Simulatie en analyse van commerciële werklasten en computersystemen

Simulating and Analyzing Commercial Workloads and Computer Systems

Frederick Ryckbosch

Promotor: prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2012 - 2013

UNIVERSITEIT
GENT

ISBN 978-90-8578-597-2
NUR 980
Wettelijk depot: D/2013/10.500/30

Dankwoord

Het moeilijkste deel aan het schrijven van een doctoraat is het schrijven van het dankwoord. Na jaren van zwoegen is het tijd om terug te kijken en stil te staan bij de personen die dit werk mogelijk gemaakt hebben. Ik zal proberen deze moeilijke taak tot een goed einde te brengen en niemand over het hoofd te zien, daarom bedank ik ten eerste de persoon die dit nu leest. Gewoon omdat je mijn scriptie leest, of je een goede collega of vriend bent, je weet zelf wel waarom ik je bedank.

Daarnaast bedank ik mijn promotor prof. Lieven Eeckhout voor de onvoorwaardelijke steun, het geven van inzichtvolle feedback en de hulp bij het schrijven van artikels en deze scriptie. Ik had me geen betere begeleiding kunnen wensen, Lieven heeft me steeds goed bijgestaan en alle mogelijkheden gegeven om dit onderzoek tot een goed einde te brengen.

Ten tweede bedank ik de leden van de examencommissie die deze scriptie gelezen, verbeterd en in vraag gesteld hebben. De vele interessante vragen hebben mijn blik op dit werk verruimd. *I would like to thank Amer Diwan especially, for hosting me during my internships at Google. It was an amazing experience being part of such an innovative and openminded company.*

In het bijzonder gaat mijn dank uit naar Stijn Polfliet voor de jarenlange samenwerking en vriendschap, bijna 10 jaar lang. En dat alles te wijten aan het feit dat Polfliet en Ryckbosch na elkaar komen in het alfabet en we dus veroordeeld waren tot het delen van een tafel tijdens het labo chemie in het eerste jaar industrieël ingenieur.

Natuurlijk wil ik ook mijn andere collega's bedanken en in het bijzonder mijn bureaugenoten Andy, Davy, Kristof, Max, Shoaib en Stijn voor de mooie tijden, de niet-vergrendelde computers, de ingepakte bureau, de Hello Kitty accessoires en zo meer.

Het Fonds Wetenschappelijk Onderzoek (FWO) voor het financieren mijn onderzoek. Het Vlaamse Supercomputer Centrum voor het beschikbaar stellen van rekenkracht en de hulp bij het opzetten van onze complexe simulatieomgevingen.

Tenslotte wil ik ook mijn familie en vrienden bedanken. Eerst en vooral mijn vriendin Els voor de steun, het geduld en het luisterend oor geduren-

de de vele ups en downs die een doctoraatsonderzoek inhouden. Verder wil ik mijn ouders bedanken. Zij hebben me de kans gegeven om te studeren en hebben me steeds gesteund in alles wat ik deed. Als laatste bedank ik mijn vrienden voor de nodige verstrooiing.

Frederick Ryckbosch

Gent, 9 mei 2013

Examencommissie

Prof. Rik Van de Walle, voorzitter
Decaan, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Koen De Bosschere, secretaris
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Bart Dhoedt
Vakgroep INTEC, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Ph.D. Amer Diwan
Google Inc., Mountain View, CA
USA

Ph.D. Ayose Falcón
Intel Barcelona Research Center
Spain

Ph.D. Wim Heirman
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Ph.D. Sofie Van Hoecke
Vakgroep ELIT
Hogeschool West-Vlaanderen

Leescommissie

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Bart Dhoedt
Vakgroep INTEC, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Ph.D. Amer Diwan
Google Inc., Mountain View, CA
USA

Ph.D. Ayose Falcón
Intel Barcelona Research Center
Spain

Ph.D. Sofie Van Hoecke
Vakgroep ELIT
Hogeschool West-Vlaanderen

Samenvatting

Gedurende de laatste jaren heeft er een grondige verschuiving plaatsgevonden in het computergebruik. Terwijl vroeger de meeste werklasten geïsoleerde programma's waren die uitvoerden op een desktopcomputer, zien we nu een verschuiving naar online-applicaties die uitvoeren op servers in een datacenter en bekeken worden met draagbare computers (zoals laptops, tablet computers en smartphones). Complexe applicaties zoals tekstverwerking en rekenbladen kunnen nu uitgevoerd worden in een Internet-browser zonder dat hiervoor nood is aan speciale software die geïnstalleerd is op de client-machine. Dit maakt het mogelijk om deze applicaties op gelijk welk moment, vanaf gelijk welke machine te gebruiken. Met de opkomst van het Internet zien we ook nieuwe soorten applicaties verschijnen zoals sociaalnetwerksites, waarmee de gebruiker kan communiceren met vrienden en familie. Er wordt verwacht dat het aantal verbonden gebruikers de komende jaren enkel zal stijgen, doordat Internetverbindingen en -apparaten steeds goedkoper worden. Een groot aantal servers is nodig om online-applicaties zoals Google of Facebook, met meer dan honderd miljoen gebruikers per dag, draaiende te houden.

De prestatie van deze online-applicaties is van het grootste belang, maar komt met een hoog prijskaartje doordat snelle servers duur zijn en veel energie verbruiken. De totale kost van een server omvat de aankoopprijs, de hoeveelheid energie die de server verbruikt tijdens zijn levensduur en de extra energie nodig voor spanningconversies en koeling.

De energie-efficiëntie en prestatie van de applicatie hangt af van zowel de software als de hardware. Computerwetenschappers gebruiken verschillende instrumenten om deze metriek te bemeten tijdens de levensduur van een server. Tijdens de ontwikkeling van een nieuw hardwareplatform, maakt men gebruik van hardware-simulatoren om de ontwerpsruimte te exploreren: verschillende ontwerpskeuzes zoals het aantal processorkernen, het type kern, de geheugenbandbreedte en de klokfrequentie worden geëvalueerd om een goede afweging te maken tussen het vermogenverbruik enerzijds en de prestatie anderzijds. Profileringssoftware en vermogenmetingen worden gebruikt om de prestatie van een bestaande applicatie op bestaande hardware te bepalen. Indien de applicatie nog niet beschikbaar is, kan men gebruik maken van synthetische werklasten om voor

een brede waaier aan applicaties het optimale hardwareplatform te bepalen. Om de daadwerkelijke prestatie van een applicatie in een productieomgeving te meten wordt typisch gebruik gemaakt van profileringssoftware en traceringssoftware.

Tijdens de levensduur van de applicaties komen de volgende uitdagingen naar boven.

1. De volgende uitdagingen komen voor tijdens het simuleren van grootschalige Internet-applicaties. Ten eerste bestaan deze applicaties uit verschillende softwarecomponenten die uitvoeren op meerdere rekenknopen. Deze softwarecomponenten omvatten besturingssystemen, virtuele machines en de volledige applicatie software. Dit verschilt sterk van traditionele benchmarks (zoals SPEC CPU) gebruikt door computerarchitectuuronderzoekers. Ten tweede gebruiken deze applicaties grote datasets, hierdoor kan het uren duren vooraleer de applicatie in een stabiele staat komt. Om dit soort werklasten te simuleren dient de simulator de volledige software stack te simuleren (inclusief het besturingssysteem) voor meerdere rekenknopen en dient de simulator snel en accuraat te zijn. Cylcusgetrouwe simulatoren zijn niet geschikt voor dit soort werklasten aangezien deze 5 tot 6 grootordes trager zijn dan de werkelijke uitvoering.

2. Software heeft een grote impact op het vermogenverbruik van een hardwareplatform. State-of-the-art werklasten zoals SPECpower gebruiken een voorgedefinieerde set werklasten om de energie-efficiëntie van een computersysteem te bepalen. Het is echter mogelijk dat de afweging tussen vermogenverbruik en prestatie verschillend is voor een applicatie met andere karakteristieken. Synthetische werklasten maken het mogelijk om een brede waaier aan applicaties te benaderen, en geven inzicht in de energie-efficiëntie voor verschillende werklasten. Daarnaast combineert SPECpower de prestatie en het vermogenverbruik van een platform in één metriek voor energie-efficiëntie, waardoor de informatie over de afweging tussen de twee metrieken verloren gaat.

3. Een andere uiterst belangrijke metriek in grootschalige Internet-applicaties is de latentie die de eindgebruiker ervaart. Een groot deel van deze latentie is bepaald door de verwerkingstijd in het datacenter en de netwerkvertraging. Wij focussen in dit werk op de verwerkingstijd in het datacenter. De meeste grootschalige Internet-applicaties hebben last van hoge latenties voor de traagste operaties. Deze latenties zijn een complex en uitdagend probleem: ze komen niet vaak voor (bijvoorbeeld in 1% van de gevallen) en worden vaak veroorzaakt door complexe interacties tussen verschillende componenten

in de software stack, mogelijks verdeeld over meerdere rekenknopen. Hierdoor is traditionele profileringssoftware niet geschikt om deze problemen te analyseren.

Grootschalige Internet-applicaties stellen hoge eisen aan zowel software als hardware. Zowel prestatie als energie-efficiëntie zijn van het grootste belang tijdens de volledige levensduur van de applicatie. Dit omvat: (i) het ontwerpen van nieuwe hardware, (ii) het evalueren van bestaande hardware en (iii) het uitvoeren van de applicatie in een productieomgeving voor echte gebruikers. Tijdens elke van deze fasen beschikt men over allerhande hulpmiddelen om de prestatie en het vermogenverbruik van deze systemen te bemeten en te analyseren. Deze thesis levert bijdragen aan elk van deze fasen. Van prestatie-evaluatie in hardware simulatoren, over energie-efficiëntiemetingen in testomgevingen tot het analyseren van hoge latenties in productieomgevingen. We leveren de volgende bijdragen in deze thesis.

Accurate simulatie Idealiter wil een computerarchitect een volledig systeem (inclusief volledige en ongewijzigde software) met hoge nauwkeurigheid in een redelijke tijdspanne simuleren. Dit is echter onmogelijk met gedetailleerde cyclusgetrouwe simulatoren aangezien deze te traag zijn. Daarom stellen we een simulatie voor op een hoger abstractieniveau, gebruik makend van een analytisch model genaamd intervalanalyse. In dit werk implementeren en integreren we dit interval-gebaseerde processormodel in de COTSon simulatorinfrastructuur.

We valideren het processormodel ten opzichte van echte hardware gebruik makend van een set micro-werklasten, meerdradige processor-intensieve werklasten en server werklasten. Dit resulteert in een gevalideerde simulatietechniek die zowel snel als accuraat is. Deze techniek maakt het mogelijk om volledige x86-werklasten, inclusief besturingssysteem en applicatiesoftware, te simuleren in een redelijke tijdspanne.

Snelle simulatie met meerdere rekenknopen Terwijl onze eerste bijdrage focust op nauwkeurige simulatie voor één machine, gaan we nu over naar het simuleren van een grote omgeving met meerdere servers. Hiervoor moet de simulatieomgeving schalen met het groot aantal servers en tegelijkertijd een goede nauwkeurigheid en snelheid aan de dag leggen.

In dit werk stellen we VSim voor, een nieuwe simulatiemethodologie voor het simuleren van systemen met meedere servers. VSim maakt gebruik van virtualisatietechnologie en tijdsvertraging. Door de prestatie van de processor, het netwerk en de harde schijven te controleren geeft VSim de software de indruk dat het op het doelsysteem uitvoert. VSim kan meerdere doelsystemen per gastheersysteem simuleren en voorziet gedistribueerde simulatie over meerdere rekenknopen voor grootschalige simulaties.

Onze experimentele resultaten tonen de nauwkeurigheid van VSim aan: de typische simulatiefout is kleiner dan 6% voor de processor, het netwerk en de harde schijf. Realistische werklasten zoals de Lucene zoekmachine en de Olio Web 2.0 werklast illustreren VSim's bruikbaarheid. Onze huidige omgeving is in staat om 5 doelsystemen per gastheersysteem te simuleren, hetgeen we aantonen via een gevalstudie met een Hadoop werklast waarin we 25 servers simuleren. Deze resultaten zijn bekomen met een vertraging van één grootorde ten opzichte van de werkelijke uitvoering.

Energie-efficiëntie en -proportionaliteit van computer systemen Vervolgens migreren we van simulatieomgevingen naar testomgevingen met echte hardware, om de energie-efficiëntie van bestaande systemen te evalueren.

De focus op energie-efficiëntie heeft gezorgd voor de opkomst van een aantal werklasten om het vermogenverbruik van computersystemen te evalueren. EEMBC bracht EnergyBench uit, SPEC heeft SPECpower en ook academici stelde verschillende werklasten voor, bijvoorbeeld JouleSort. Gebruik makend van de beschikbare vermogen- en prestatiemetingen voor een brede set van commerciële hardwareplatforms, analyseren we hoe de energie-proportionaliteit geëvolueerd is gedurende de laatste drie jaar. Daarnaast evalueren we hoe goed SPECpower energie-proportionaliteit kwantificieert en bestuderen we hoeveel energie bespaard kan worden door servers meer energie-proportioneel te maken.

Een grote beperking van EnergyBench, SPECpower, JouleSort, enz. is dat de resultaten afhankelijk zijn van de specifieke werklast, en dus weinig inzicht geven tijdens het vergelijken van de energie-efficiëntie van twee systemen. Hiervoor stellen we SWEEP (Synthetische Werklasten voor Energie-Efficiëntie en Prestatie-evaluatie) voor, een raamwerk dat synthetische werklasten genereert met specifieke karakteristieken. We gebruiken SWEEP om een brede waaier van synthetische werklasten te genereren met een verschillende instructiemix, instructie-niveauparallellisme, geheugentoegangspatronen en I/O-intensiteit. We gebruiken SWEEP om de energie-efficiëntie van commerciële computersystemen te evalueren voor een grote waaier aan werklasten en leren hoe de energie-efficiëntie gekoppeld is met de werklastenkarakteristieken.

Daarnaast introduceren we het energieprestatiediagram, een nieuwe methode om de afweging tussen energie en prestatie voor te stellen. Dit diagram geeft meer inzicht dan de traditionele EDP en ED^2P metrieken.

Analyseren van hoge latenties De laatste bijdrage gebruikt data van een echte productieomgeving, bestaande uit traces afkomstig van server in een datacenter dat echte klanten bedient.

Terwijl profileringssoftware van onschatbare waarde is voor het oplos-

sen van prestatieproblemen die veel voorkomen, heeft ze weinig nut bij het oplossen van prestatieproblemen die voorkomen in een heel klein deel van de gevallen (bijvoorbeeld de 1% traagste operaties). Deze operaties hebben een grote impact op de kost en gebruikerservaring van grootschalige Internet-applicaties. Aangezien deze hoge latentie vaak veroorzaakt worden door complexe interacties tussen verschillende softwarecomponenten, is het nodig om fijnkorrelige traces te analyseren om de oorzaak van het probleem te achterhalen.

Het analyseren van traces is een moeizame activiteit aangezien men moet redeneren over lange aaneenschakelingen van gebeurtenissen en dit vaak een diepe domeinkennis over de gebeurtenissen vergt. Om deze problemen te verhelpen stellen we TPA voor, een taal gebaseerd op temporele logica, voor het analyseren van traces. Dit stelt een expert in staat zijn domeinkennis te noteren, waarna het systeem alle overeenkomstige gebeurtenissen zoekt en nuttige informatie extraheert over de gevonden gebeurtenissen. We tonen aan dat ons systeem schaalt en ons in staat stelt om latentie-problemen op te sporen in een echte productieomgeving, namelijk Google's GMail.

Summary

In recent years we have witnessed a profound shift in computing: computer workloads are shifting from isolated programs running on a personal computer to online applications running on servers in datacenters, accessed by portable computers (like laptops, tablets or smartphones). Complex applications like text processing or spreadsheets are now available through the Internet and no longer require dedicated software to be installed on the client machine, making it possible to use these applications at anytime and anywhere. The Internet has also leveraged the rise of new types of applications such as social networks through which people connect and interact with friends and family. In future years the number of connected users is expected to grow, as devices and Internet connections are becoming cheaper. Huge amounts of servers are required to serve Web applications like Google or Facebook with hundreds of millions of daily users.

Performance is a primordial concern for a lot of online applications. This quest for performance comes at a price however, as using fast power-hungry servers can be very expensive. The total cost of ownership of a server is defined by the purchase price, the amount of power the server consumes and the additional power required to run the server, which includes power conversion and cooling.

The energy efficiency and performance of the application depends on both software and hardware. Computer scientists and engineers use various tools to determine these metrics at the different stages of the lifetime of a server. During the development of a new hardware platform, simulators are used to explore the design space: different design options such as core count, core type, cache sizes, memory bandwidth, clock frequency, etc. are evaluated to determine a good power versus performance trade-off. While evaluating existing hardware, profilers and power measurements are used to determine the performance of a given application. In case the target application is not available, benchmarks are used to evaluate the hardware platform. Synthetic benchmarks can be used to cover a wide range of applications, which enables evaluating existing hardware for different kinds of target applications. On production systems, profiling and tracing are typically used to measure the actual performance of the application.

Throughout the lifetime of an application the following challenges arise.

1. Simulating new hardware for large-scale Internet applications has a number of hurdles. These applications consist of various software components running on multiple nodes, and include complex software stacks running commercial operating systems, virtual machines along with an entire application stack. These workloads are very different from the traditional benchmarks (e.g., SPEC CPU) used by computer architecture researchers. Because of the large data sets processed by these applications, it can take hours before the application reaches steady state behavior. To simulate these applications, a hardware simulator has to be able to simulate an entire software stack (including the operating system) for multiple nodes and be fast and accurate. Typical cycle-accurate simulators introduce a slowdown of at least 5 to 6 orders of magnitude compared to native execution, making them infeasible for simulating these applications.

2. The power consumed by a hardware platform highly depends on the software running on the platform. State-of-the-art benchmarks like SPECpower use a predefined workload to determine the energy efficiency of a platform. It is however possible that the trade-offs are different for an application with different characteristics. Synthetic benchmarks can be used to cover a broad spectrum of applications and provide more insight into the energy efficiency of the system for different types of benchmarks. SPECpower calculates a single metric for energy efficiency, which combines both the execution time and the energy usage. While there is a trade-off between these two metrics, SPEC provides an arbitrary metric that lumps these two metrics together and does not show the performance versus energy trade-off.

3. Another key metric in large-scale Internet applications is the latency perceived by the end user. This latency is determined by the processing time in the datacenter and the network latency. In this work, we focus on the processing time in the datacenter. Most large-scale Internet applications suffer from long-tail latencies. These latencies have some challenging properties: they do not occur often (e.g., in 1% of the cases) and are often caused by complex interactions between many components in the software stack, possibly across multiple nodes. Because of this, traditional profilers are inadequate for analyzing these problems.

Running a modern online application places heavy requirements on both software and hardware. Performance and energy efficiency are of great importance throughout the whole lifetime of the application, which includes: (i) designing new hardware, (ii) evaluating available hardware and

(iii) running the application in production for real customers. During each of these phases different tools are used to measure and analyze the performance and energy usage of these systems. This thesis makes contributions for each of these phases. From performance evaluation in hardware simulators, over energy-efficiency measurements in test environments to long latency analysis in production environments. More specifically, this dissertation makes the following contributions.

Accurate Full-system Simulation Ideally, a computer architect would want to simulate an entire system with high accuracy in a reasonable amount of time while running complete and unmodified software stacks. However, common practice of detailed cycle-accurate processor simulation is becoming infeasible because of being too slow. We therefore propose CPU timing simulation at a higher level of abstraction, and we present an approach for doing so, using an analytical model called interval analysis. We implement and integrate this interval-based CPU timing model in the COTSon full-system simulation infrastructure.

We extensively validate the timing model against real hardware using a set of micro-benchmarks, (multi-threaded) CPU-intensive benchmarks and a server workload. The end result is a validated simulation approach that is both accurate and fast, and in addition can run full-system x86 workloads, including commercial operating systems and entire software stacks in an affordable amount of time.

Fast Multi-Node Simulation While the first contribution focuses on accurately simulating a single machine, we subsequently scale the simulation to a high-end setup involving multiple servers. The simulation environment therefore needs the ability to scale out to a large number of server nodes while attaining good accuracy and reasonable simulation speeds.

We propose VSim, a novel simulation methodology for multi-server systems. VSim leverages virtualization technology for simulating a target system on a host system. VSim controls CPU, network and disk performance on the host, and it gives the illusion to the software stack to run on a target system through time dilation. VSim can simulate multiple targets per host and employs a distributed simulation scheme across multiple hosts for simulations at scale.

Our experimental results demonstrate VSim's accuracy: average errors are below 6% for CPU, disk and network performance. Real-life workloads involving the Lucene search engine and the Olio Web 2.0 benchmark illustrate VSim's utility and accuracy (average error of 3.2%). Our current setup can simulate up to five target servers per host, and we provide a Hadoop workload case study in which we simulate 25 servers. These simulation results are obtained at a simulation slowdown of one order of

magnitude compared to native hardware execution.

Computer System Energy Efficiency and Energy Proportionality We subsequently migrate from simulation environments to test environments with real hardware, and we more specifically evaluate the energy efficiency of existing systems.

The focus on energy efficiency has led to a number of power benchmarking methods recently. For example, EEMBC released EnergyBench and SPEC released SPECpower to quantify a system's energy efficiency; also academics have proposed power benchmarks, such as JouleSort. Using the power and performance number from a broad set of commercial machines, we analyze how energy-proportionality has evolved over the past three years. We evaluate to what extent SPECpower quantifies energy-proportionality, and we study how much total energy can be saved by making servers even more energy-proportional.

A major limitation for EnergyBench, SPECpower, JouleSort, etc. is that they are tied to a specific benchmark, and hence, they provide limited insight with respect to why one system may be more energy-efficient than another. We propose therefore SWEEP (Synthetic Workloads for Energy Efficiency and Performance evaluation), a framework for generating synthetic workloads with specific behavioral characteristics. We employ SWEEP to generate a wide range of synthetic workloads while varying the instruction mix, ILP, memory access patterns, and I/O-intensiveness; and we use SWEEP to evaluate the energy efficiency of commercial computer systems across the workload space and learn about how the energy efficiency of a computer system is tied to its workload's characteristics.

We also present the Energy-Delay Diagram (EDD), a novel method for visualizing energy efficiency. The EDD clearly illustrates the energy versus performance trade-off, and provides more intuitive insight than the traditionally used EDP and ED^2P metrics.

Analyzing Long-Tail Latencies The last contribution uses data from a production environment, consisting of traces from servers in a datacenter running Web applications with real-world traffic from customers.

While profiling is invaluable for debugging performance problems that affect the common case, it is of little help in tracking performance problems that affect the slowest 1% of the operations (i.e., long-tail latencies). For Web service providers, these long-tail latencies affect both the cost of the service and the user experience. Since interactions between operations are often responsible for long-tail latency, we must analyze fine-grained traces to investigate their cause.

Unfortunately, analyzing traces is difficult because one needs to reason over long chains of events and because this reasoning often requires

significant domain knowledge about what the event sequences mean. We therefore propose TPA, a language based on linear-temporal logic, for analyzing traces. Given these formulas, our system searches through traces to find matches for these formulas and extracts relevant information from the matches. We demonstrate that our system is scalable and enables us to investigate long-tail performance problems in a real production environment, namely Google's GMail service.

Contents

List of Tables

List of Figures

List of Abbreviations

AMD	Advanced Micro Devices
CEP	Complex event processing
CPU	Central processing unit
DMA	Direct memory access
DRAM	Dynamic random-access memory
ED2P	Energy delay squared product
EDD	Energy delay diagram
EDP	Energy delay product
EEMBC	Embedded Microprocessor Benchmark Consortium
FPGA	Field-programmable gate array
FTP	File transfer protocol
GNU	GNU's not Unix!
HDD	Hard disk drive
HPET	High Precision Event Timer
HTML	HyperText Markup Language
I/O	Input/output
ILP	Instruction-level parallelism
IP	Internet protocol
IPC	Instructions per clock cycle
ISA	Instruction-set architecture
JRE	Java runtime environment
JVM	Java virtual machine
KIPS	Thousands of instructions per second
L1	Level-1 Cache
L2	Level-2 Cache
L3	Level-3 Cache
LTL	Linear temporal logic
MIPS	Millions of instructions per second
MLP	Memory-level parallelism
NIC	Network interface card
OLTP	Online transaction processing
OS	Operating system
PARSEC	Princeton Application Repository for Shared-Memory Computers

PDES	Parallel Discrete-Event Simulation
PUE	Power usage efficiency
QoS	Quality of service
RISC	Reduced instruction set computing
RMS	Recognition, Mining and Synthesis
RPC	Remote procedure call
SMT	Simultaneous multithreading
SPEC	Standard Performance Evaluation Corporation
SQL	Structured query language
SSD	Solid-state drive
SSE	Streaming SIMD Extensions
SUT	System under test
TCO	Total cost of ownership
TDP	Thermal design power
TLB	Translation lookaside buffer
VM	Virtual machine
VMM	Virtual machine monitor
XML	Extensible Markup Language

Chapter 1

Introduction

In recent years we have witnessed a profound shift in computing: computer workloads are shifting from isolated programs running on a personal computer to online applications running on servers in datacenters, accessed by portable computers (like laptops, tablets or smartphones). Complex applications like text processing or spreadsheets are now available through the Internet and no longer require dedicated software to be installed on the client machine, making it possible to use these applications at anytime and anywhere. The Internet has also leveraged the rise of new types of applications such as social networks through which people connect and interact with friends and family. In future years the number of connected users is expected to grow, as devices and Internet connections are becoming cheaper. Huge amounts of servers are required to serve Web applications like Google search or Facebook with hundreds of millions of daily users. Barroso et al. [1] view this ensemble of servers as one big computer: a warehouse-scale computer. This scale poses new challenges but also provides new opportunities for optimization. Since the largest part of the workload is running in a datacenter controlled by a single company, optimizing both the datacenter software and hardware can have a huge impact on the performance, energy efficiency and total cost of ownership.

End-user latency is a primordial concern for a lot of online applications. Research at Amazon [2], a popular online shop, has shown that an increase in page load time of 100 ms decreases sales by 1%. Experiments at Google [3], the online search engine, have shown that increasing the number of search results from 10 to 30 increased the latency by 500 ms and caused a 20% drop in ad revenue. The performance of the application in the datacenter defines a large portion of the end-user latency.

This quest for performance comes at a price however, as using fast power-hungry servers can be very expensive. The total cost of ownership of a server is defined by the purchase price, the amount of power the server

consumes and the additional power required to run the server, which includes power conversion and cooling. Additional cost factors of a datacenter include facility, personnel, maintenance, etc. The power usage effectiveness (PUE) metric [4] is defined as Total Facility Energy divided by the IT Equipment Energy and abstracts the additional power into a factor on the amount of power that the server consumes. In a state-of-the-art datacenter facility, a PUE of 1.06 can be achieved [5]: this means that the additional power is 6% on top of the power consumed by the servers. Designing the most appropriate hardware for a specific application can lower the required power, and thus reduce the total cost of ownership of the hardware.

The energy efficiency and performance of the application depends on both software and hardware. Computer scientists use various tools to determine these metrics at different stages during the lifetime of a server. During the development of a new hardware platform, simulators are used to explore the design space: different design options such as core count, core type, cache sizes, memory bandwidth, clock frequency, etc. are evaluated to determine a good power versus performance trade-off. While evaluating existing hardware, profilers and power measurements are used to determine the performance of a given application. In case the target application is not available, benchmarks are used to evaluate the hardware platform. Synthetic benchmarks can be used to cover a wide range of applications, which enables evaluating existing hardware for different kinds of target applications. On production systems, profiling and tracing are typically used to measure the actual performance of the application.

1.1 Thesis topic

Running a modern online application places heavy requirements on both software and hardware. Performance and energy efficiency are of great importance throughout the whole lifetime of the application, which includes: (i) designing new hardware, (ii) evaluating available hardware and (iii) running the application in production for real customers. During each of these phases different tools are used to measure and analyze the performance and energy usage of these systems. This thesis makes contributions for each of these phases: from performance evaluation in hardware simulators, over energy-efficiency measurements in test environments to long latency analysis in production environments. The remainder of this chapter introduces the key challenges addressed and contributions made in this thesis.

1.2 Key challenges

1.2.1 Evaluating new hardware

Simulating new hardware for large-scale Internet applications faces a number of hurdles. These applications consist of various software components running on multiple nodes, and include complex software stacks running commercial operating systems, virtual machines along with an entire application stack. These workloads are very different from the traditional benchmarks (e.g., SPEC CPU) used by computer architecture researchers. Because of complex applications layer caching and the large data sets processed by these applications, it can take hours before the application reaches steady-state behavior. To simulate these applications, a hardware simulator has to be able to simulate an entire software stack (including the operating system) for multiple nodes, be fast and accurate. Typical cycle-accurate simulators introduce a slowdown of at least 5 to 6 orders of magnitude compared to native execution, making them infeasible for simulating these applications.

1.2.2 Evaluating existing hardware

The power consumed by a hardware platform highly depends on the software running on the platform. State-of-the-art benchmarks like SPECpower use a predefined set of benchmarks to determine the energy efficiency of a platform. It is however possible that the trade-offs are different for an application with different characteristics. Synthetic benchmarks can be used to cover a broad spectrum of applications and provide more insight into the energy efficiency of the system for different types of benchmarks. SPECpower calculates a single metric for energy efficiency, which combines both the execution time and the energy usage. While there is a trade-off between these two metrics, SPEC provides an arbitrary metric that lumps these two metrics together and not show the performance versus energy trade-off.

1.2.3 Analyzing production systems

As mentioned before, another key metric in large-scale Internet applications is the latency perceived by the end user. This latency is determined by the processing time in the datacenter and the network latency. In this work, we focus on the processing time in the datacenter. Most large-scale Internet applications suffer from long-tail latencies. These latencies have some challenging properties: they do not occur often (e.g., in 1% of the cases) and are often caused by complex interactions between many compo-

nents in the software stack, possibly across multiple nodes. Because of this, traditional profilers are inadequate for analyzing these problems.

1.3 Contributions

This dissertations makes contributions in various fields: hardware simulation, energy efficiency measurements and analyzing infrequent performance problems.

Contribution 1: Accurate Full-system Simulation

Ideally, a computer architect would want to simulate an entire system with high accuracy in a reasonable amount of time while running complete and unmodified software stacks. However, common practice of detailed cycle-accurate processor simulation is becoming infeasible because of being too slow. We therefore propose CPU timing simulation at a higher level of abstraction, and we present an approach for doing so, using an analytical model called interval analysis [6, 7]. We implement and integrate this interval-based CPU timing model in the COTSon full-system simulation infrastructure [8].

We extensively validate the timing model against real hardware using a set of micro-benchmarks, (multi-threaded) CPU intensive benchmarks and a server workload. The end result is a validated simulation approach that is both accurate and fast, and in addition can run full-system x86 workloads, including commercial operating systems and entire software stacks in an affordable amount of time.

This work on accurate simulation has been published in:

> Frederick Ryckbosch, Stijn Polfliet and Lieven Eeckhout, "Fast, Accurate, and Validated Full-System Software Simulation of x86 Hardware", In *IEEE Micro*, Vol. 30, No. 3, 46-56, November 2010.

Contribution 2: Fast Multi-Node Simulation

While the first contribution focuses on accurately simulating a single machine, we subsequently scale the simulation to a high-end setup involving multiple servers. The simulation environment therefore needs the ability to scale out to a large number of server nodes while attaining good accuracy and reasonable simulation speeds.

We propose VSim, a novel simulation methodology for multi-server systems. VSim leverages virtualization technology for simulating a target

system on a host system. VSim controls CPU, network and disk performance on the host, and it gives the illusion to the software stack to run on a target system through time dilation. VSim can simulate multiple targets per host and employs a distributed simulation scheme across multiple hosts for simulations at scale.

Our experimental results demonstrate VSim's accuracy: average errors are below 6% for CPU, disk and network performance. Real-life workloads involving the Lucene search engine and the Olio Web 2.0 benchmark illustrate VSim's utility and accuracy (average error of 3.2%). Our current setup can simulate up to five target servers per host, and we provide a Hadoop workload case study in which we simulate 25 servers. These simulation results are obtained at a simulation slowdown of one order of magnitude compared to native hardware execution.

This work on high-level simulation has been published in:

> Frederick Ryckbosch, Stijn Polfliet and Lieven Eeckhout, "VSim: Simulating multi-server setups at near native hardware speed", In *ACM Transactions on Architecture and Code Optimization*, 52:1-52:20, January 2012.

Contribution 3: Computer System Energy Efficiency and Energy Proportionality

We subsequently migrate from simulation environments to test environments with real hardware, and we more specifically evaluate the energy efficiency of existing systems.

The focus on energy efficiency has led to a number of power benchmarking methods recently. For example, EEMBC released EnergyBench [9] and SPEC released SPECpower [10] to quantify a system's energy efficiency; also academics have proposed power benchmarks, such as JouleSort [11]. Using the power and performance number from a broad set of commercial machines, we analyze how energy-proportionality has evolved over the past three years. We evaluate to what extent SPECpower quantifies energy-proportionality, and we study how much total energy can be saved by making servers even more energy-proportional.

A major limitation for EnergyBench, SPECpower, JouleSort, etc. is that they are tied to a specific benchmark, and hence, they provide limited insight with respect to why one system may be more energy-efficient than another. We propose therefore SWEEP (Synthetic Workloads for Energy Efficiency and Performance evaluation), a framework for generating synthetic workloads with specific behavioral characteristics. We employ SWEEP to generate a wide range of synthetic workloads while varying the instruction mix, ILP, memory access patterns, and I/O-intensiveness; and we use

SWEEP to evaluate the energy efficiency of commercial computer systems across the workload space and learn about how the energy efficiency of a computer system is tied to its workload's characteristics.

We also present the Energy-Delay Diagram (EDD), a novel method for visualizing energy efficiency. The EDD clearly illustrates the energy versus performance trade-off, and provides more intuitive insight than the traditionally used EDP and ED^2P metrics.

This work on a evaluating energy efficiency has been published in:

> Frederick Ryckbosch, Stijn Polfliet and Lieven Eeckhout, "Trends in Server Energy Proportionality", In *IEEE Computer 44(9)*, pages 69-72, September 2011.
>
> Kristof Du Bois, Tim Schaeps, Stijn Polfliet, Frederick Ryckbosch and Lieven Eeckhout, "SWEEP: Evaluating Computer System Energy Efficiency using Synthetic Workloads", In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 159-166, January 2011.

Contribution 4: Analyzing Long-Tail Latencies

The last contribution uses data from a production environment, consisting of traces from servers in a datacenter running Web applications with real-world traffic from customers.

While profiling is invaluable for debugging performance problems that affect the common case, it is of little help in tracking performance problems that affect the slowest 1% of the operations (i.e., long-tail latencies). For Web service providers, these long-tail latencies affect both the cost of the service and the user experience. Since interactions between operations are often responsible for long-tail latency, we must analyze fine-grained traces to investigate their cause.

Unfortunately, analyzing traces is difficult because one needs to reason over long chains of events and because this reasoning often requires significant domain knowledge about what the event sequences mean. We therefore propose TPA, a language based on linear-temporal logic, for analyzing traces. Given these formulas, our system searches through traces to find matches for these formulas and extracts relevant information from the matches. We demonstrate that our system is scalable and enables us to investigate long-tail performance problems in a real production environment, namely Google's GMail service.

This work on analyzing long-tail latencies is submitted to:

> Frederick Ryckbosch and Amer Diwan, "Analyzing performance

traces using temporal formulas", Submitted to *the Journal of Software: Practice and Experience*, 2013.

1.4 Overview

The remainder of this dissertation is organized as follows.

We present an accurate full-system simulator in Chapter 2. We validate this simulator against commercial hardware, explore accuracy versus speed trade-offs, and show its use for real-life workloads.

In Chapter 3, we focus on simulating multi-node systems, using the high-level VSim simulator. Disk and network are also simulated besides the processor. After validating the simulation results, we discuss two case studies with workloads running on tens of servers.

The energy proportionality of contemporary servers is analyzed in Chapter 4. We extend upon this work for measuring energy efficiency of commercial hardware in Chapter 5. We use synthetic benchmarks to emulate a wide range of applications and show that the energy efficiency for a hardware platform very much depends on the software and its characteristics.

Chapter 6 introduces the TPA language. This work was done during an internship at Google, Inc. and was used to analyze performance traces from the GMail production servers.

Chapter 7 concludes this dissertation with a summary of the work and outlook towards future work.

Chapter 2

Accurate Full-System Simulation

This chapter presents a fast and accurate interval-based CPU timing model implemented in the COTSon full-system simulation infrastructure. Validation against real x86 hardware demonstrates the accuracy of the timing model. The end result is a software simulator that faithfully simulates x86 hardware running complete software stacks at a speed in the tens of MIPS range.

2.1 Introduction

Architectural simulation is a challenging problem in contemporary computer architecture research and development, for two reasons. For one, contemporary processors integrate billions of transistors on a single chip, implement multiple cores along with on-chip peripherals, and are complex pieces of engineering. In addition, modern software stacks are becoming more and more complex as well, and include commercial operating systems, virtual machines along with an entire application stack, which is very different from the workloads traditionally considered in computer architecture research (e.g., SPEC CPU). Ideally, a computer architect would want to simulate an entire system with high accuracy in a reasonable amount of time while running complete and unmodified software stacks.

However, common practice of detailed cycle-accurate processor simulation is becoming infeasible because of being too slow. Moreover, one may argue that cycle-accurate simulation may not be called for in many practical studies. Many design trade-offs need to be made at the system level for which the slow speed and high level of detail of cycle-accurate simulation only gets in the way. We therefore propose CPU timing simulation at a higher level of abstraction, and we present an approach for doing so, using

an analytical model called interval analysis [6,7]. The analytical model is mechanistic in nature — it is built on first principles starting from a deep understanding of the system that is to be modeled — and analyzes a program's miss events (such as cache misses, tlb misses and branch mispredictions) as well as its dependence structure to estimate CPU performance. We implement and integrate this interval-based CPU timing model in the COTSon full-system simulation infrastructure [8]. We extensively validate the timing model against real hardware using a set of micro-benchmarks, (multi-threaded) CPU-intensive benchmarks and a server workload. The end result is a validated simulation approach that is both accurate and fast, is relatively easy to implement, and in addition can run full-system x86 workloads, including commercial operating systems and entire software stacks, and system devices such as network cards and disks in an affordable amount of time.

We make the following contributions in this work:

- We propose a validated interval-based analytical timing model for a superscalar out-of-order x86 processor core and integrate this model into HP Labs' COTSon simulator. We describe the development and validation process, which demonstrates that the interval-based CPU timer is relatively easy to implement. The timing model consists of around 1,500 lines of C code (including comments), and was written in about 1 engineer-month. The validation and fine-tuning process against real hardware took another 3 engineer-months.

- We validate the interval-based CPU timer against real hardware, namely an AMD Opteron 2350 quad-core server processor [12], using a set of micro-benchmarks and synthetic kernels. The average error compared to real hardware is 9.8% for the micro-benchmarks, and 18.6% for a set of CPU-intensive benchmarks including SPECjbb2005, H.264 video decoding and encoding, bio-informatics, and multi-threaded PARSEC benchmarks. We obtain similarly accurate results for a non-trivial Web 2.0 search engine server workload: 7.0% average error for response time and 12.7% for throughput across a range of concurrent clients.

- We study the trade-off in speed versus accuracy when enabling sampling in the COTSon simulator with the new interval-based CPU timer. The end result is a full-system software simulator that faithfully simulates x86 hardware at a speed in the tens of MIPS range: one particular sampling strategy achieves a speed of 37 MIPS and an average error of 23.1% for the CPU-intensive workloads. Given its high speeds, we believe our simulator may be useful to drive software optimizations and hardware/software co-optimization.

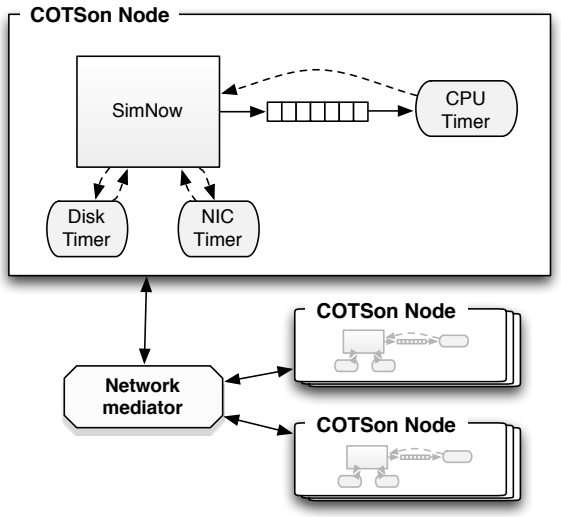

Figure 2.1: COTSon's architecture.

2.2 COTSon

COTSon [8] is an open-source simulator framework developed by HP Labs. Its goal is to provide a fast evaluation vehicle for current and future computing systems, covering both entire software stacks and hardware modules including processors and system devices such as network cards and disks. COTSon targets cluster-level systems consisting of multiple multicore processor nodes interconnected through a network, i.e., it targets both scale-up (i.e., multicore and manycore processor simulation) as well as scale-out (i.e., simulation of a multinode cluster).

Figure 2.1 shows the organization of the COTSon simulator. COTSon uses the AMD SimNow full-system simulator to functionally simulate each node in the cluster. AMD's SimNow can simulate x86 and x86_64 processors, and uses dynamic compilation and code caching techniques to speed up simulation. SimNow is around $10\times$ slower than native hardware execution, and is capable of booting a system with an unmodified operating system and can execute any complex application. Each COTSon node further consists of timing models for the disks, network card interface and the CPU (i.e., processor and memory). The various COTSon nodes are interconnected through a network mediator.

The timing models in each COTSon node communicate with the functional simulator through event queues. These event queues are either synchronous (for communicating with the disk and network timing models) or asynchronous (for communicating with the CPU timing model).

Synchronous event queues need an immediate response from the timing model upon a request from the functional simulator. Asynchronous event queues on the other hand decouple the generation of events by the functional simulator and their processing by the timing models. Asynchronous event queues implement a unique timing feedback mechanism which periodically adjusts the speed of the functional simulator to reflect the timing estimates provided by the timing models. This is called functional-directed simulation and approximates timing behavior more accurately than purely trace-driven or functional-first simulation (while being faster than timing-directed execution-driven simulation). Timing feedback allows for better approximating time-dependent behavior (e.g., synchronization, OS scheduling, networking, etc.), which is important for real-life workloads in terms of load balancing, quality-of-service, etc.

COTSon simulates multicore processors by serializing the functional simulation of the various cores. Each core is allowed to run for some fixed amount of time in the functional simulator, and when all cores have reached the same point in time (the simulation window), COTSon sends the various instruction streams to the timing models. Hence, it is the functional simulator that determines functional behavior, not the timing simulator; for example, the functional simulator determines which thread acquires the lock for entering a critical section. The timing models then determine the progress for each of the cores, which in turn adjust the speed of the functional simulator through timing feedback; for example, if the timing model determines that a core achieves an instruction throughput (IPC) that is twice as high as for another core, the functional simulator will simulate twice as many instructions for that one core compared to the other in the next simulation window. The feedback mechanism aims at limiting the divergence of the functional simulator with respect to the timing simulator.

The open-source version of COTSon comes with two CPU timing models, `timer0` and `timer1`, which are timing models for an in-order and out-of-order processor, respectively. These CPU timing models are fairly simple, and are primarily designed for tutorial purposes and not to provide realistic levels of accuracy. In particular, `timer1` operates as follows. It stalls the front-end pipeline upon an I-cache/TLB miss and branch misprediction. Loads have priority over stores, and can be issued to memory as long as there are memory ports available. There are many aspects that this timer does not model. It does not model miss event overlaps, hardware prefetching, break-up of macro-operations into micro-operations; neither does it model the impact of instruction execution latencies and inter-instruction dependencies (i.e., the impact of the critical path is not modeled). The average error compared to real hardware for `timer1` for our set of micro-benchmarks and CPU-intensive benchmarks equals 42.4% and

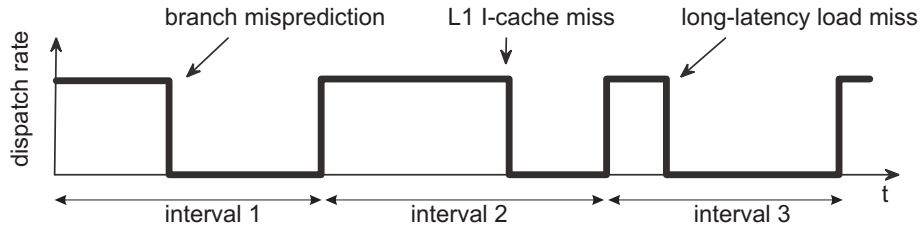

Figure 2.2: Interval analysis analyzes performance on an interval basis determined by disruptive miss events.

31.8%, respectively. The interval-based CPU timing model, which we describe next, achieves substantially higher levels of accuracy. In this work, we use the existing COTSon network and disk timers.

2.3 Interval simulation

The CPU timing model presented in this work is based on a recently proposed analytical performance model for superscalar out-of-order processors, called interval analysis [6]. The model is mechanistic in nature which means that it is built on first principles: the performance model is derived in a bottom-up fashion, starting from a basic understanding of the mechanics of a contemporary processor.

Interval analysis partitions a program's execution time into intervals separated by disruptive miss events such as cache misses, TLB misses, branch mispredictions and serializing instructions. This is illustrated in Figure 2.2, which shows the number of dispatched instructions on the vertical axis versus time on the horizontal axis. Under optimal conditions, i.e., in the absence of miss events, the processor sustains a level of performance more-or-less equal to its pipeline front-end dispatch width — we refer to dispatch as the point of entering the instructions from the front-end pipeline into the reorder buffer and issue queues. However, miss events disrupt the smooth streaming of instructions through the dispatch stage. By dividing execution time into intervals, one can analyze the performance behavior of the intervals individually. In particular, one can, based on the type of interval (the miss event that terminates it), determine the performance penalty per miss event:

- The penalty for an I-cache/TLB miss equals its miss delay.
- The penalty for a branch misprediction equals the branch resolution time (number of cycles between the branch entering the reorder buffer and issue queue, and its resolution) plus the front-end pipeline depth.

- The penalty per long-latency load miss (i.e., a last-level cache/TLB load miss) is approximated by its miss delay (memory access time). Multiple independent load misses may overlap their execution and expose memory-level parallelism (MLP).
- The penalty for a serializing instruction equals the reorder buffer drain time.

The smooth streaming of instructions between miss events, at a rate close to the designed dispatch width, may not always be achieved. Low-ILP applications may exhibit long chains of dependent instructions, L1 data cache misses and long-latency functional unit instructions (divide, multiply, floating-point operations, etc.), or store instructions, which may cause a resource (e.g., reorder buffer, issue queue, physical register file, write buffer, etc.) to fill up. A resource stall may thus cause dispatch to eventually stall for a number of cycles. This is modeled in interval modeling through an ILP model that computes the critical path over a window of instructions while keeping track of the inter-instruction dependencies and instruction execution latencies. The intuition is that the window (reorder buffer) cannot slide over the dynamic instruction stream any faster than dictated by the critical path. The effective dispatch rate is then computed through Little's law (reorder buffer size divided by critical path length), capped by the designed dispatch width.

Interval analysis also provides good insight into how miss events overlap with each other. For example, the penalty due to an I-cache miss that follows a long-latency load miss is hidden underneath the long-latency load penalty. Similarly, the penalty for a mispredicted branch that follows a long-latency load in the dynamic instruction stream on which it does not depend, is completely hidden underneath the penalty due to the long-latency load. If on the other hand, the mispredicted branch depends on the long-latency load, then both penalties serialize.

By employing interval modeling, one can build architecture simulators that model the target machine at a higher level of abstraction. Interval simulation was recently proposed by Genbrugge et al. [7]. The basic idea of interval simulation is to replace the cycle-accurate core-level timing model by the interval model. The core-level interval model then interacts with the branch predictor and memory subsystem simulators to derive the miss events and (possibly) their latencies. The interval model then estimates how many cycles it takes to execute each interval. This includes analyzing the amount of ILP to determine the effective dispatch rate between miss events, as well as estimating how many cycles it takes to resolve a mispredicted branch and to drain the reorder buffer on a serializing instruction. Finally, the model also estimates the amount of overlap between miss events in order to do an accurate accounting in terms of their penalties. In

other words, the interval model estimates a core's overall progress based on timing estimates of each individual interval: the miss events are determined by simulating the branch predictor and memory subsystem — the miss events determine the intervals — and the timing for each interval is estimated through the interval model. The key benefits of interval simulation are that it is easy to implement and runs substantially faster than cycle-accurate simulation, while maintaining good accuracy. Genbrugge et al. validated the interval simulator against the M5 simulator [13] which implements the Alpha RISC ISA: they achieved an average error of 4.6% and a $10\times$ simulation speedup compared to detailed simulation while running full-system multi-threaded workloads.

2.4 Accurate x86 CPU timing model

There are three major goals that we set ourselves to achieve in this work.

- *Validation against real hardware.* Although the earlier work described in the previous section demonstrated the accuracy of interval modeling and simulation, its validation was done compared to an academic simulator. This is a good first step, however, it is unclear how accurate the model is against real hardware. Prior work in simulator validation has shown that it is extremely difficult to validate an academic simulator against real hardware [14]. This raises the question whether a model that has been validated against a simulator is close to real hardware.
- *Validation for the prevalent x86 ISA.* The earlier work in interval modeling and simulation — alike many other modeling and simulation efforts in computer architecture — focus on a RISC ISA that is relatively easy to handle, namely Alpha. The question can be raised whether this is sufficient given the prevalence of the x86 (and x86_64) ISAs in contemporary computer systems. Moreover, given that we target the simulation infrastructure at simulating computer systems running real and unmodified software stacks, x86 is the ISA of choice.
- *Fast and accurate simulation environment for running unmodified commercial workloads.* Finally and foremost, we wanted an accurate, fast and easy-to-implement simulator that can run unmodified commercial full-system workloads at scale in an affordable amount of time. Although the COTSon simulation infrastructure fulfills most of these requirements — it is fast and can run unmodified complex workloads — the available CPU timing models are simple tutorial models.

The key idea that initiated this work was to integrate the interval model as a CPU timing model into the COTSon infrastructure. This would solve

all three of the above challenges. It enables validation for the x86 ISA; it enables validation against real hardware (given the predominance of x86 hardware); and it would possibly improve the accuracy of the COTSon infrastructure. The end result of this project is that we achieved all three goals: the interval model based CPU timing model significantly improves the accuracy of the COTSon simulation infrastructure compared to real hardware running complex x86 workloads.

2.4.1 Modeling

Because the interval model is relatively easy to implement — which is one of its key assets — its integration in COTSon was done quickly. We implemented the interval model as a novel CPU timing model in COTSon in about one engineer-month. This includes the interval model itself along with a number of particularities that relate to x86 architectures. Subsequently, validation was done against real hardware, which took another three engineer-months. This validation process was done against an AMD Opteron server system (see the experimental setup section for more details) and revealed several opportunities for improving the model. Building a validated interval-based CPU timing model took four engineer-months in total.

Compared to the interval model proposed in earlier work [7], the novel interval-based CPU timing model includes the following novel features.

- *Micro-operations.* The interval-based CPU timing model breaks x86 instructions (macro-operations) into RISC-like micro-operations; this break-up is done in a generic way: an x86 instruction is broken up in one or more load micro-ops, followed by an arithmetic operation and one or more store micro-ops. Our current implementation does not include macro-op nor micro-op fusion, although this could be added easily.

- *x86 disassembly.* We had to integrate an x86 disassembler as part of the novel CPU timing model to enable micro-op formation and to determine an instruction's type as well as its input and output operands. x86 disassembly also involves register assignment and dependence analysis to create data dependencies between the micro-ops. Note that the integration of a disassembler into the timing model is a result of the fact that the COTSon simulator leverages AMD's proprietary SimNow functional simulator which does not expose the instruction type and operands to COTSon — if SimNow would communicate disassembly information to COTSon, there would be no need to integrate a disassembler in the timing model.

- *Hardware prefetching.* All modern high-end processors implement some form of hardware prefetching in order to hide memory access latencies. Prior versions of the interval simulator did not include hardware prefetching though. On par with the AMD Opteron processor [12] that we validate against, the novel interval-based CPU timing model implements hardware prefetching at multiple levels of the memory hierarchies, namely at the core-level L1 data cache — called the core prefetcher — and at the L3 cache level — called the DRAM prefetcher. The core prefetcher is instruction pointer based; the DRAM prefetcher initiates prefetches based on the observed L3 cache access patterns. Both prefetchers are stride-based.

- *Overlapping miss events.* Interval analysis assumes that only off-chip memory accesses, i.e., last-level L3 cache misses, cause the reorder buffer to fill up and stall dispatch; other misses such as L2 misses that hit in L3 are assumed to be hidden through out-of-order execution. We found this to be an invalid assumption for the real hardware we validated against. Therefore, we consider L3 hits as another source of miss events and we apply the overlap algorithm to L3 hits accordingly, i.e., dispatch is assumed to block on an L3 hit, and independent miss events further down the dynamic instruction stream that make it into the reorder buffer simultaneously with the L3 hit, may (partially) overlap with this L3 hit.

- *Latency tuning.* Interval analysis uses instruction latencies to determine the length of the critical data dependence path through the program, which in its turn is important to determine the effective dispatch rate in the absence of miss events. Unfortunately, instruction execution latencies are poorly documented. We therefore considered synthetically generated kernels to determine instruction latencies. We used this procedure for determining the latencies of a number of instruction types, such as integer divide and multiply operations, floating-point operations, SSE operations, etc.

2.4.2 Validation against real hardware

The validation process against real hardware revealed many opportunities for improving the interval-based CPU timing model. Figure 2.3 shows the progress during the validation process. The vertical axis shows the absolute error between the simulator and the real hardware for a set of micro-benchmarks (details follow later). For each intermediate version of the timing model we show the average absolute error (diamond) as well as its standard deviation (error bar).

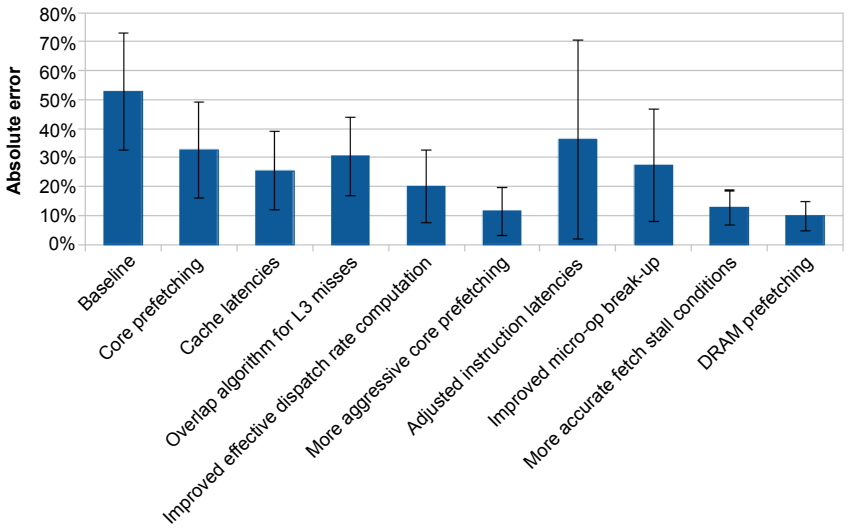

Figure 2.3: Validation process using the micro-benchmarks and synthetically generated kernels.

1. *Baseline* This implementation is based on the original interval simulation implementation, the most important change is the move from the Alpha ISA to the x86_64 ISA. Which includes the break-up of x86 instructions into micro-operations.

2. *Core prefetching* In this step we add a core prefetcher: this prefetcher is executed on every load, independent of whether they miss and hit in the L1 data cache. The core prefetcher contains a number of stride-based prefetchers, indexed by the instruction pointer of the instruction that does the load.

3. *Cache latencies* Adjusted the latencies of a cache access to 1 cycle for the level 1 cache, 8 cycles for level 2 cache and 14 cycles for level 3 cache.

4. *Overlap algorithm for L3 misses* The original interval simulation work looks at 2 levels of caching. Since we are validating against a processor with 3 levels of caching, we add logic to take the overlapping effects of the L3 into account. While the original model only looks at MLP effects for main memory accesses, we also determine the MLP effects for L3 accesses.

5. *Improved effective dispatch rate computation* The original model did not take the instruction fetch into account as a possible bottleneck. In this step we introduced a resource window that keeps track of the number of instructions that are fetched in each cycle. When not enough

resources are available to execute the instruction fetch, it is postponed to the next cycle, effectively modeling the instruction fetch as a possible bottleneck.

6. *More aggressive core prefetching* The core prefetcher now fetches the next 4 addresses instead of only 1 address.

7. *Adjusted instruction latencies* After step 6, the average latency drops to 11.5% but the mul microbenchmark still shows an error of 24.8%. Since the mul microbenchmark is relatively simple, the error can only be explained by an incorrect latency for the multiply instruction. After changing the latencies for the load, multiply and store micro-operations, the error for the mul benchmark decreased at the cost of an increase in error for the other benchmarks. We are however confident that this is a step in the right direction, because we are now able to model the mul microbenchmark with higher accuracy.

8. *Improved micro-op break-up* We improve the micro-op break-up by separating registers that are used for calculating memory addresses from registers that are used to store data. Moreover we also seperate the registers that contain memory addresses that are read from registers that contain memory addresses that are written. The read addresses are required before any calculation can start, while the write addresses are required after the execution of the instruction. Before this step the execution of an instruction was waiting for all memory addresses to become available, this means that now the execution of the instruction can start before the destination memory address is known.

9. *More accurate fetch stall conditions* In this step we make sure that an instruction cannot be executed before it is fetched, this constrained was not explicitly checked before.

10. *DRAM prefetching* In the last step, we added a second prefetcher at the memory controller level: only the L3 cache misses are provided to this prefetcher. This prefetcher is also a stride-based prefetcher.

The steps above brought the average error down to 9.8% with a max error of 19.8% (see the rightmost point in the graph).

2.5 Experimental setup

Hardware platform. The hardware platform that we validate against is an AMD Opteron 2350 quad-core processor machine [12]. It implements AMD's K10 microarchitecture in a 65 nm technology at 2 GHz; each core

Micro-benchmarks		
bsearch	Binary search through a sorted array	10.5
dijkstra	Compute shortest path using Dijkstra's algorithm	8.8
div	Compute the sum of quotients	0.8
dl1	Compute the sum across an array that fits in L1	12.1
fp	Chain of dependent floating-point operations	1.8
memory	Compute sum across a large array that does not fit in L3	8.0
mul	Compute sum of products	7.2
qsort	Quicksort algorithm on an array of random values	3.9
Compute-intensive benchmarks: PARSEC, BioPerf, MediaBench 2, SPECjbb		
blackscholes	Option pricing with Black-Scholes PDE	1.5
bodytrack	Body tracking of a person	4.5
freqmine	Frequent itemset mining	1.9
ferret	Content similarity search server	11.7
streamcluster	Online clustering of an input stream	4.8
raytrace	Real-time raytracing	1.9
swaptions	Pricing of a portfolio of swaptions	4.1
blastn	Identification of similar nucleotide sequences in a database	0.1
blastp	Identification of similar protein sequences in a database	1.5
ce	Finds structural similarities between pairs of proteins	10.9
h264dec	H.264 video decoding	3.1
h264enc	H.264 video encoding	6.1
specjbb2005	SPEC's benchmark for evaluating the Java server performance. The benchmark models a wholesale company, with warehouses that serve a number of districts. Customers initiate new orders or request the status of existing orders.	4.3
Server workload		
nutch	Nutch is an open source web-search framework which builds on Lucene Java, adding web-specifics, such as a crawler, a link-graph database, parsers for HTML and other document formats. It uses a client machine to generate search requests to the Nutch webserver and measures response time and throughput at the client side.	100 - 200

Table 2.1: Benchmarks used in this study. The rightmost column show the number of instructions executed (in billions of instructions).

is a 3-wide superscalar out-of-order architecture with a 72-entry reorder buffer. The L1 caches are 64 KB in size; further it implements a per-core 512 KB L2 cache, and a shared 2 MB L3 cache along with an on-chip memory controller.

We repeated our real hardware measurements 15 times and we report average performance numbers along with its confidence intervals. The measurements were done on an idle machine, and time was measured using the Linux `time` command.

Benchmarks. Table 2.1 lists the benchmarks used in this study. We make a distinction between micro-benchmarks, compute-intensive benchmarks and server benchmarks. The micro-benchmarks stress specific aspects of the architecture such as floating-point units, divide, core prefetching, DRAM prefetching, etc. The compute-intensive benchmarks is a set of benchmarks taken from a variety of sources such as PARSEC [15], BioP-

erf [16], MediaBench II [17] and SPECjbb2005; the PARSEC benchmarks are multi-threaded and model Recognition, Mining and Synthesis (RMS) workloads. This set of benchmarks covers classes of workloads such as data analytics, presentation, multimedia, gaming, etc. which are likely candidates to run in (future) computer systems. Finally, the server benchmark is a Web 2.0 search engine workload called Nutch: a client sends search requests to the Nutch server and measures the response time and throughput at the client side.

2.6 Evaluation: accuracy versus speed

The evaluation of the interval-based CPU timer within COTSon is done in a number of steps. We first focus on accuracy, and consider the micro-benchmarks and the compute-intensive benchmarks. Subsequently, we focus on the speed versus accuracy trade-off while employing sampling.

Accuracy: micro-benchmarks and CPU-intensive benchmarks. Figure 2.4 compares the relative error for the interval-based timer against real hardware execution using the micro-benchmarks when reporting simulated time in seconds. The average absolute error equals 9.8%. The interval-based CPU timer is also accurate for the compute-intensive benchmarks, see Figure 2.5. The average absolute error for the interval-based timer equals 18.6% (max error of 41%). As mentioned before, the PARSEC benchmarks are multi-threaded workloads, and we run up to four threads, because the AMD Opteron machine that we compare against is a quad-core processor. As we increase core count, we also increase the number of threads that co-execute, and these co-executing threads affect each other's performance through synchronization as well as through shared resources (e.g., L3 cache, off-chip bandwidth, main memory). Interval-based CPU modeling captures these interactions well. Note though that AMD's SimNow serializes the functional simulation of cores, which may lead to different behavior seen during functional simulation than what would be seen in a timing-directed simulator or on real hardware, e.g., a spin lock loop may be iterated a different number of times in COTSon than on real hardware — this is especially a concern for workloads with high contention locks. Functional-directed simulation as implemented in COTSon addresses this concern to some extent. The error numbers reported here include this inaccuracy. One solution may be to more tightly couple the speed of the functional simulator on the one side and the timing simulator on the other side, however, doing so without compromising simulation speed too much is an orthogonal issue that falls outside the scope of this work.

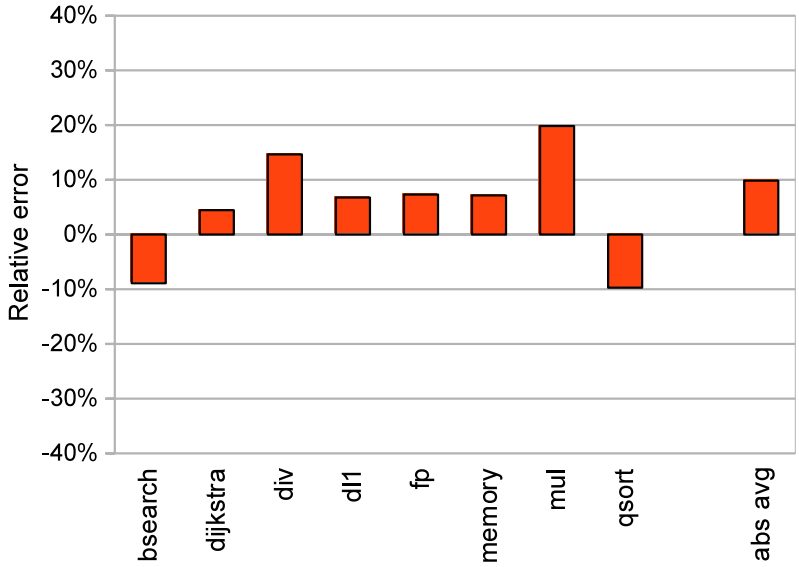

Figure 2.4: Validation of the interval-based CPU timing model using micro-benchmarks.

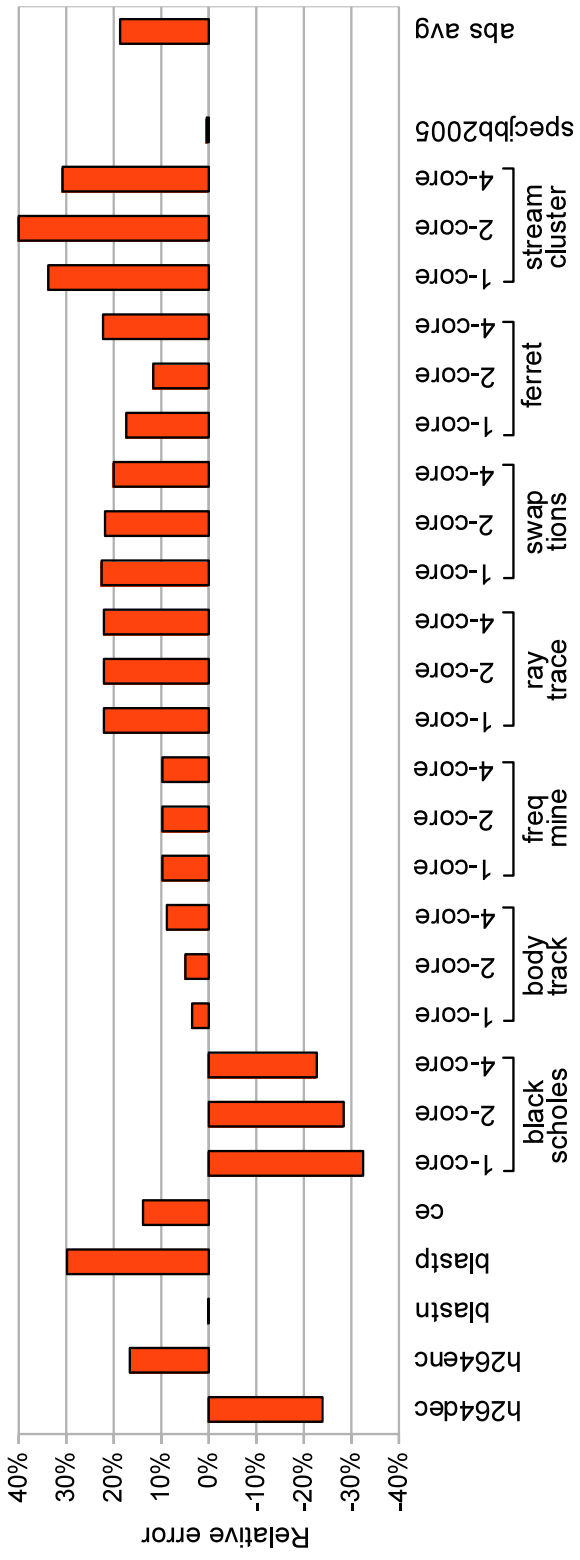

Figure 2.5: Validation of the interval-based CPU timing model using a suite of compute-intensive benchmarks from BioPerf, MediaBench II, PARSEC, and SPECjbb2005.

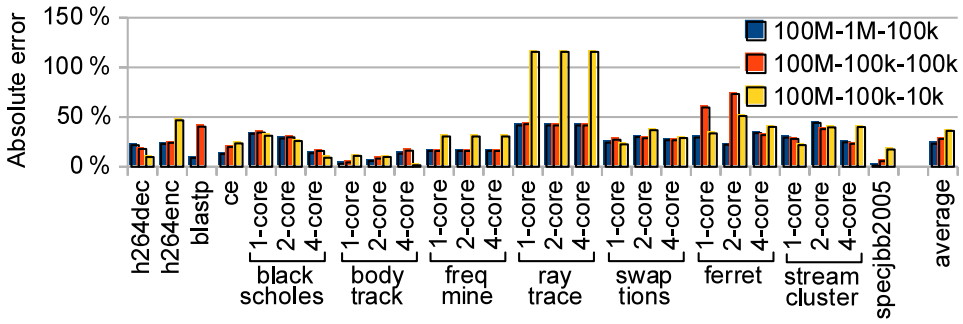

Figure 2.6: Accuracy for three sampling strategies: (i) 1M warming and 100K instruction sampling units, (ii) 100K warming and 100K instruction sampling units, and (iii) 100K warming and 10K instruction sampling units; there are 100M instructions between the sampling units for all three strategies.

Sampling: trading off speed versus accuracy. Running complex full-system workloads — which is our ultimate goal — requires that very long running workloads can be simulated in a reasonable amount of time. Our interval-based CPU timing model achieves 350 KIPS (which is 38% slower than the COTSon CPU timer which runs at 570 KIPS). Although this is a reasonable simulation speed, it is not fast enough to simulate complex workloads in an affordable amount of time. A well-founded technique for speeding up simulation is sampling [18–20]. The idea behind sampling is to simulate only a small fraction of the entire dynamic instruction stream in detail and then extrapolate, i.e., by taking small sampling units randomly or periodically one can get an accurate picture of the entire execution. Since only a small fraction is simulated in detail, substantial speedups are obtained. Figure 2.6 shows the accuracy for three sampling scenarios — we explored more strategies during our work but they are not shown here to improve readability — (i) 1M warming and 100K instruction sampling units, (ii) 100K warming and 100K instruction sampling units, and (iii) 100K warming and 10K instruction sampling units; there are 100M instructions between the sampling units for all three strategies. Accuracy improves with increasing sampling unit size and more warming. The 1M warming and 100K sampling unit scenario achieves an average error of 23.1% and a simulation speed of 37 MIPS. Figure 2.7 shows the trade-off in accuracy versus speed, and considers a number of sampling strategies. We find the 100K sampling strategy (with one sampling unit every 100M instructions and 1M instructions of warming) to be a good trade-off in speed versus accuracy, and we use this sampling strategy further.

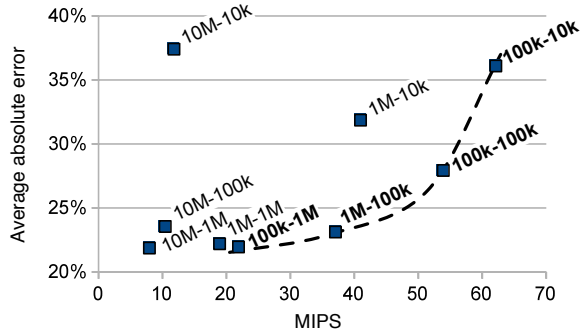

Figure 2.7: Speed versus accuracy trade-off: the Pareto front is formed through the dashed line. A sampling strategy A-B means A instructions for warming and B instructions for the sampling unit. All sampling strategies assume 100M instructions between sampling units.

2.7 Case study: Server workload

We now consider a more complex server workload, namely a Web 2.0 search engine application that is based on the Nutch platform. Nutch is open-source web-search software, built on Lucene Java while adding various Web specifics such as crawling, HTML parsing, a link-graph database, etc. Our benchmark consists of a server holding the search database and a variable number of clients that submit requests to the server; the server is run on one COTSon simulation node, and the clients are run on another one. Figures 2.8 and 2.9 show the response time and throughput on the client side, respectively, for both the real hardware and COTSon (which uses the interval-based CPU timer). The simulation is within 7.0% and 12.7% on average for response time and throughput, respectively. It is interesting to observe that throughput increases up to 100 concurrent clients with only modest increase in response time. Throughput decreases dramatically past 140 clients, with a highly variable transition phase between 100 and 140 clients. Software simulation captures this trend very well.

The real power of software simulation is that it allows for exploring the microarchitecture and its effect on overall performance. Figure 2.10 shows a case study in which we consider three L3 cache sizes: 1 MB, 8 MB and 32 MB. The response time for the Nutch benchmark decreases with increasing cache sizes, and 1 MB seems to be sufficient for limited levels of concurrency, whereas an 8 MB cache is clearly beneficial for larger numbers of concurrent clients, and a 32 MB cache brings no further improvement.

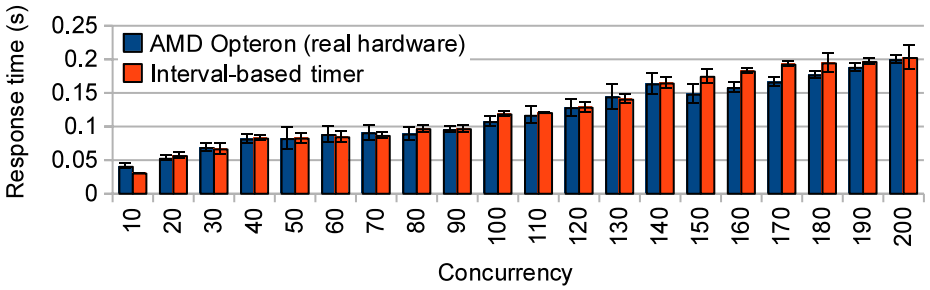

Figure 2.8: Response time for the nutch benchmark.

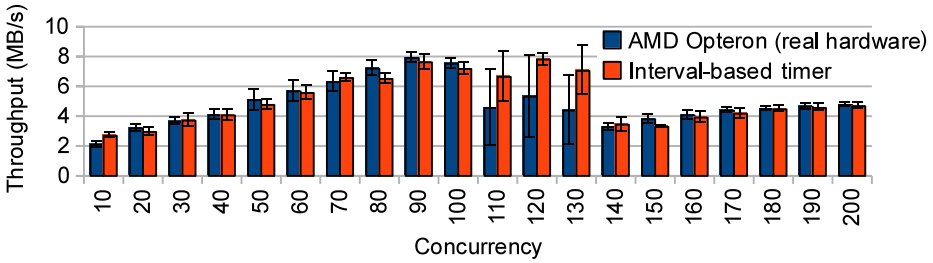

Figure 2.9: Throughput for the nutch benchmark.

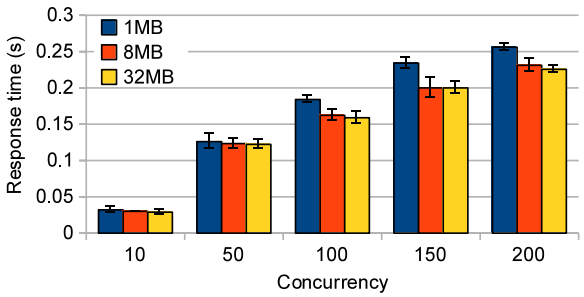

Figure 2.10: Microarchitecture study varying the cache size for the nutch benchmark.

2.8 Related work

Execution-driven simulation. Mauer et al. [21] present a useful taxonomy for execution-driven simulation. Functional-first simulation lets a functional simulator feed a trace of instructions into a timing simulator, which may lead to loss in accuracy along mispredicted paths and when simulating multi-threaded workloads. In timing-directed simulation, functional simulation is driven by the timing simulator, i.e., the timing simulator directs the functional simulator when to change architecture state. Timing-first simulation lets the timing simulator run ahead with the functional simulator as a checker. COTSon [8] implements a so-called functional-directed simulation paradigm: the functional simulator can run ahead of the timing simulator, however its speed is periodically adjusted by the timing simulator — functional-directed simulation could be viewed as middle ground between functional-first and timing-directed simulation.

FPGA-accelerated simulation. Various research groups are focusing on FPGA-accelerated simulation [22]. An FPGA-accelerated simulator exploits fine-grain parallelism and achieves high simulation speeds on the order of tens of MIPS. A limitation of FPGA acceleration is that it may expand simulator development time because it requires modeling the target architecture in a hardware description language such as Verilog, VHDL or Bluespec. The software simulation approach presented in this work falls within the same speed range, while being much easier to develop — four engineer-months for the validated CPU timing model within the COTSon infrastructure.

Simulator validation. Simulator validation is a non-trivial and tedious endeavor. Desikan et al. [14] validated the detailed cycle-level `sim-alpha` simulator against the Alpha 21264 processor. They were able to improve the simulator to be within 2% compared to the real hardware for a set of micro-benchmarks. However, when running real SPEC CPU benchmarks, the average error was around 20%. The interval-based CPU timer proposed in this work is a simulation model at a much higher level of abstraction compared to `sim-alpha`, yet it is equally accurate for the CPU-intensive workloads.

Sampling. Wenisch et al. [23] propose Flex Points as a way to increase simulator performance for multi-threaded commercial workloads. Flex Points are full simulator checkpoints (containing microarchitectural state in caches, branch predictors, etc.) for each region of the benchmark that needs to be simulated. This eliminates the need for functional warming

between the regions of interest.

Falcón et al. [24] propose a dynamic sampling approach that uses detailed simulation for a small portion of each phase of the simulated application and uses virtualization to skip the remainder of the phase. Virtualization statistics such as code cache invalidations and code exceptions are used to identify phase changes. This allows the authors to detect phase changes during fast forwarding and thus eliminates the need for a profiling run. This technique significantly speeds up simulations while introducing only a small error. This technique can be used in conjunction with our interval simulator.

More recent work. After this paper was published, our group kept on working on fast but accurate architectural simulation, which resulted in the Sniper [25] simulator. Sniper is built on the Graphite parallel simulation infrastructure [26] and uses a validated interval simulation model. Unlike our work which focused on commercial workloads and systems, Sniper is targeted towards high-performance scientific computing workloads and systems.

2.9 Conclusion

Simulation is an invaluable tool for contemporary system design. This chapter presented a fast, accurate and validated interval-based CPU timing model that was easily implemented and integrated in the COTSon full-system simulation infrastructure. The validation was done against an AMD Opteron machine and showed that the simulator is within 20% compared to real hardware for a set of compute-intensive workloads; the simulator was also shown to be accurate for a non-trivial server workload. The end result is a full-system software simulator that can run unmodified x86 workloads at a speed in the tens of MIPS range.

Chapter 3

Fast Multi-Node Simulation

Whereas the previous chapter focused on simulating a single server, we now present VSim, a novel simulation methodology for multi-server systems. VSim leverages virtualization technology for simulating a target system on a host system. VSim controls CPU, network and disk performance on the host, and it gives the illusion of running on the target system to the software stack through time dilation. VSim can simulate multiple targets per host and employs a distributed simulation scheme across multiple hosts for simulations at scale.

3.1 Introduction

Simulation is a vital tool in contemporary experimental research and development, both in hardware and software design. Because simulation is so widely used for different purposes, there exists a whole range of simulation techniques. Emulation or functional simulation is at one end of the spectrum and models the functional aspects of a computer system only. The simplest form of emulation is interpretation in which the simulator interprets one instruction at a time [27]. Dynamic translation can significantly speed up simulation by caching previously translated instructions [28, 29]. Emulation does not produce timing information, and is most useful to determine whether a design is functionally correct. At the other end of the spectrum, cycle-accurate simulation models both the timing and the functional aspects of a computer system at a very high level of detail [13, 23, 30, 31]. Hence, accuracy is excellent but simulation speed is problematic — industry-grade simulators typically incur a slowdown of at least 5 or 6 orders of magnitude compared to native hardware execution.

Simulating servers, multi-server setups and datacenters is particularly challenging, for a number of reasons. For one, simulating user-level benchmarks, such as SPEC CPU, is unlikely to be representative for server workloads. Instead, one needs the ability to simulate a complete software stack

including the operating system, (process) virtual machines, middleware, application software, etc. In addition and related to this, simulating a multi-server setup not only involves modeling CPU performance but also network and disk activity. Further, the simulation technique needs to be scalable: it requires the ability to scale out and simulate a large number of nodes, and, if the goal is to use simulation to make design decisions, it is crucial that one can simulate a future target system on a contemporary host. Moreover, a developer may not have (frequent) access to a large number of host servers to run simulations. This implies the ability to simulate large target systems on smaller host systems. Finally, simulation speed is obviously a key property. Cycle-accurate simulation is clearly not the appropriate approach for simulating multi-server systems. Instead, we need fast simulation techniques that produce meaningful performance data in a reasonable amount of time. Although we want high simulation speed we do not want to compromise on accuracy too much. The purpose of simulation is to steer decision making, hence the simulation data does not necessarily need to be cycle-accurate, yet it should be accurate enough to make good design decisions.

This chapter proposes VSim, a novel full-system simulation methodology that leverages virtualization technology to simulate multi-server setups. VSim consists of a system virtual machine that runs on a host server and controls CPU, network and disk performance as perceived by the software, i.e., the software is given the illusion to run on a target system with performance properties that differ (significantly) from the host. Virtualization also enables simulating multiple target servers per host by running target servers as guest virtual machines. Distributed simulation across multiple hosts enables simulation at scale.

The key contribution made in VSim is to enable full-system simulation (including CPU, disk and network) using time dilation: VSim filters timer interrupts delivered to the guests, making the time perceived by the guest (simulated time) a fraction of the physical time on the host (simulation time or wall clock time). At the same time, VSim schedules the guests at native hardware speed on the host. This not only enables simulating multiple target servers per host but it also enables controlling the performance perceived by the guests. For example, by scheduling a guest for a longer period of time on the host, one can model a higher performance target server, i.e., the simulated target server gets more work done per unit of (simulated) time. Next to controlling CPU performance, VSim also controls network and disk bandwidth and latency: all network and disk requests pass through VSim which controls their bandwidth and latency in simulated time.

Our prototype implementation in Oracle's VirtualBox demonstrates the potential of the technique. We provide results showing that VSim can accu-

rately model CPU, network and disk performance. In particular, we model both high-end (AMD Opteron) and low-end (Intel Atom) target CPUs on a mid-scale host system with an average error of 2.0%; we model 1 Gbit and 100 Mbit target networks with an average error of 4.9%; and we model SSD (Solid State Drive) and HDD (Hard Disk Drive) disks with average errors of 3.3% and 5.5%, respectively (all errors are calculated versus real hardware). When put together, VSim can simulate complete computer systems running complete software stacks with good accuracy. We demonstrate this through a client-server setup running the Lucene search engine which involves simulating a client and a server running an operating system, Java virtual machine, network protocol stack and disk I/O (average error of 3.2%). We also consider case studies involving more complex setups, including the Olio Web 2.0 benchmark running a Web server, file server and database server, as well as a 25-server Hadoop workload setup. VSim achieves good accuracy while incurring a simulation slowdown of one order of magnitude only compared to native hardware execution. Moreover, VSim can simulate multiple target servers per host server (up to five target servers per host server in our current setup with an average error below 5%).

Given that VSim can simulate large systems on smaller systems with good accuracy at good speed while running complete software stacks, we believe that VSim is a promising approach towards simulating systems at scale. Moreover, VSim is complementary to other simulation paradigms, i.e., whereas cycle-accurate simulation remains an important tool in the computer designer's toolbox for making detailed microarchitecture design decisions, VSim is a more appropriate tool for making system-level design decisions in multi-server setups. Finally, both software developers and system architects can benefit from VSim. Software developers can use VSim to evaluate software optimizations and tune their software before deployment on a target system. System architects can use VSim to evaluate server design trade-offs and make system-level design decisions. Service providers can potentially use VSim to find performance bottlenecks in existing systems: they can deploy an existing software stack on a VSim simulation model and then vary both hardware and software parameters to identify performance issues.

This chapter is organized as follows. Section 3.2 presents and describes the VSim simulation paradigm in detail. Section 3.3 then elaborates on the experimental setup which we use to evaluate VSim in Section 3.4. Finally, we describe related work (Section 3.5), and we conclude (Section 3.6).

3.2 VSim

VSim models a target computer system by running a modified virtualization layer on top of a host computer system. The virtualization layer gives the illusion to the software stack to run on the target computer system. The target computer system could be an existing or a future computer system that runs a complete software stack. The simulation is done through a virtualization layer, VSim, which is run on top of a host, and the target software stack is run on top of the virtualization layer. The host system is likely to be different from the target system, e.g., it is potentially smaller in size (e.g., has fewer servers, fewer disks, etc.). VSim is a modified system virtual machine that controls the performance observed by the software stack, i.e., the software stack is given the illusion to run on the target system.

3.2.1 General concept

VSim is built on the notion of the timer interrupt. In order to understand how this is done, we first briefly revisit the purpose of the timer interrupt in a computer system. The hardware periodically generates timer interrupts and the timer interrupt handler keeps track of time by counting the number of timer interrupts. System software (operating system, hypervisor, virtual machine monitor) uses the timer interrupts to drive the scheduling of processes on the hardware. In other words, the software stack observes physical time, i.e., the software stack receives and handles all timer interrupts delivered by the hardware.

The key idea of time dilation is to let VSim manipulate the timer interrupt mechanism on the host system such that the software stack is given the illusion to run on a different system, namely the target system. VSim receives the timer interrupts and sees simulation time (i.e., physical time on the host), and does the following three actions: (i) it limits the amount of simulation time given to the software stack (i.e., it schedules the software stack for only a fraction of the simulation time), (ii) it measures the overhead due to virtualization, and (iii) it filters the timer interrupts given to the software stack. Other timing sources for the operating system in the host system, such as HPET and RDTSC, are dilated in the same way as the timer interrupts. The combination of these actions makes the software stack believe that it runs as fast as on the target system because it performs the same amount of work over the same simulated time period.

Figure 3.1 illustrates this for simulating a single server. Whereas on the target system the software stack receives all the physical timer interrupts, on the simulated system, the software stack receives only a fraction of the timer interrupts. In this example, VSim filters one out of two timer interrupts and it schedules the target for a *simulation window* per *simulation*

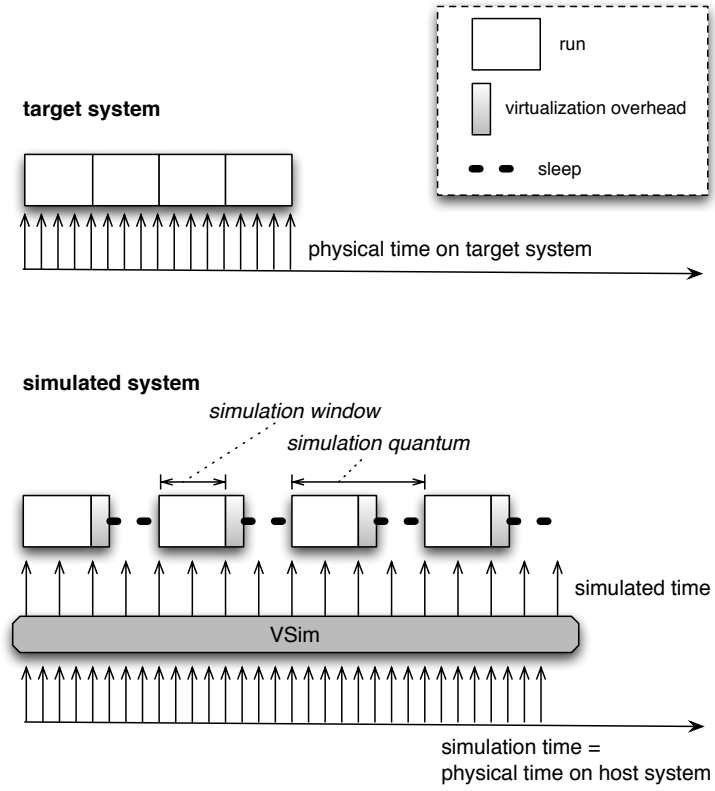

Figure 3.1: Simulating one server in VSim.

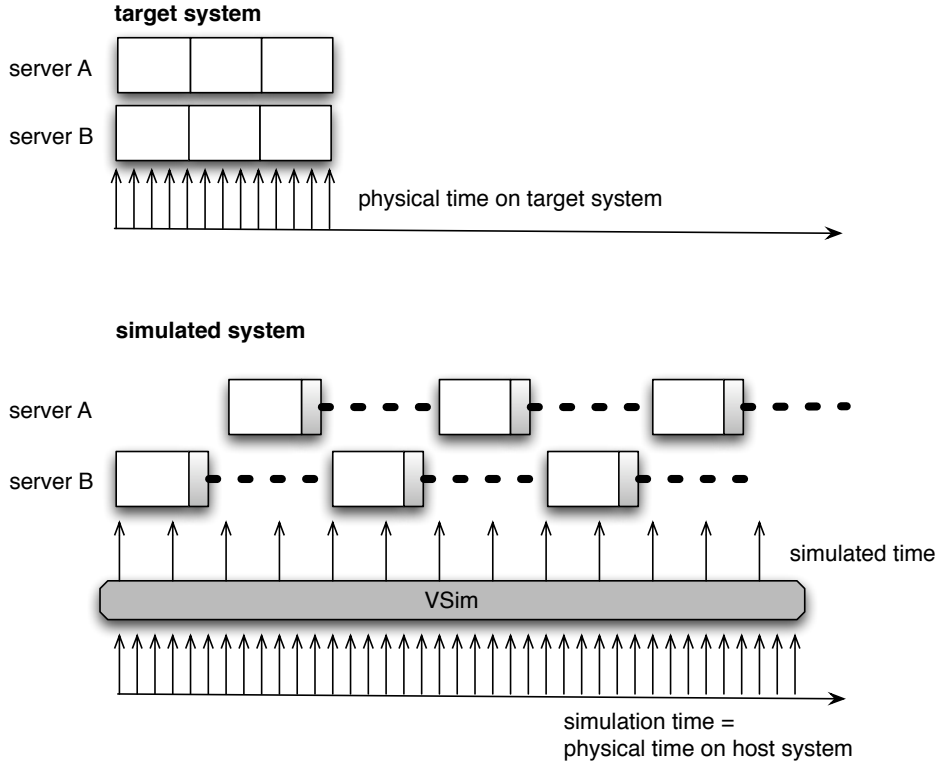

Figure 3.2: Simulating multiple servers on a single host server in VSim.

quantum. In other words, VSim schedules the target on the host during the simulation window and then puts the target to sleep until the next simulation quantum. VSim schedules the software stack on the host system such that the simulated system gets as much work done on average per unit of time as the target system: e.g., the physical target system gets a unit of work done in 4 (physical) time units (see Figure 3.1 on the top) and the simulated target system also gets a unit of work done in 4 (simulated) time units (see Figure 3.1 at the bottom). When the software stack is scheduled to run on the host machine it runs at the host's native speed, but the time progress given to the software stack is the simulated time, not the physical time on the host machine. The virtualization layer also measures its own overhead which it does not include when computing simulated time. The slowdown during simulation is the ratio of the simulated time divided by the simulation time, or 2× in this example.

We can employ this approach for simulating multiple target servers on a single host server; Figure 3.2 illustrates this for two servers. Referring back to Figure 3.1, because of the virtualization layer overhead, we cannot run two target servers on a single host at a simulation slowdown of 2×

compared to native target execution, i.e., we cannot schedule units of work for both target systems within a simulation quantum. We therefore target a simulation slowdown of 3× here in this example and filter one timer interrupt every three physical timer interrupts, see Figure 3.2, i.e., we prolong the simulation quantum.

These general concepts immediately illustrate the power of VSim. It enables simulating multiple servers running complete software stacks at high speed on a single server. By manipulating the simulated time, one can model different server configurations and study its impact on overall system performance. For example, one can model a faster CPU by filtering out more timer interrupts (i.e., the software stack is given the illusion that it gets more work done per unit of time); or, one can model different network latency and bandwidth properties by controlling the latency and bandwidth of the packets as they traverse the VSim virtualization layer; or, one can model different disk configurations by controlling the latency and bandwidth of disk accesses by the VSim virtualization layer. We now describe in more detail how we model CPU, network and disk performance in VSim.

3.2.2 CPU simulation

Modeling CPU performance can be done following two approaches, see also Figure 3.3. The first approach keeps the timer interrupt filtering constant (e.g., VSim filters one out of two timer interrupts) and adjusts the amount of work done per simulation quantum. The second approach, which is dual to the first one, is to keep the amount of work done per simulation quantum constant and to adjust the timer interrupt filtering by VSim. In Figure 3.3, both approaches model a target system with a target performance of 1.5 relative to the baseline system shown in Figure 3.1. Approach #1 performs 6 units of work per 4 simulated time units, and approach #2 performs 4.5 units of work per 3 simulated time units.

Both approaches are in essence the same, however, we opt for approach #1 in VSim because if one is to simulate a heterogeneous system consisting of multiple servers with different relative speeds, approach #1 naturally keeps simulated time synchronized among the simulated servers. In approach #1, heterogeneity is modeled by executing different units of work per simulation quantum, i.e., VSim would execute more units of work per simulation quantum for a high-performance server as compared to a less powerful server. In approach #2 on the other hand, VSim would keep the amount of work done constant per server per simulation quantum and would filter out more timer interrupts for the high-end server than for the low-end server. As a result, time would progress faster for the high-end server than for the low-end server, and thus, time would diverge on the

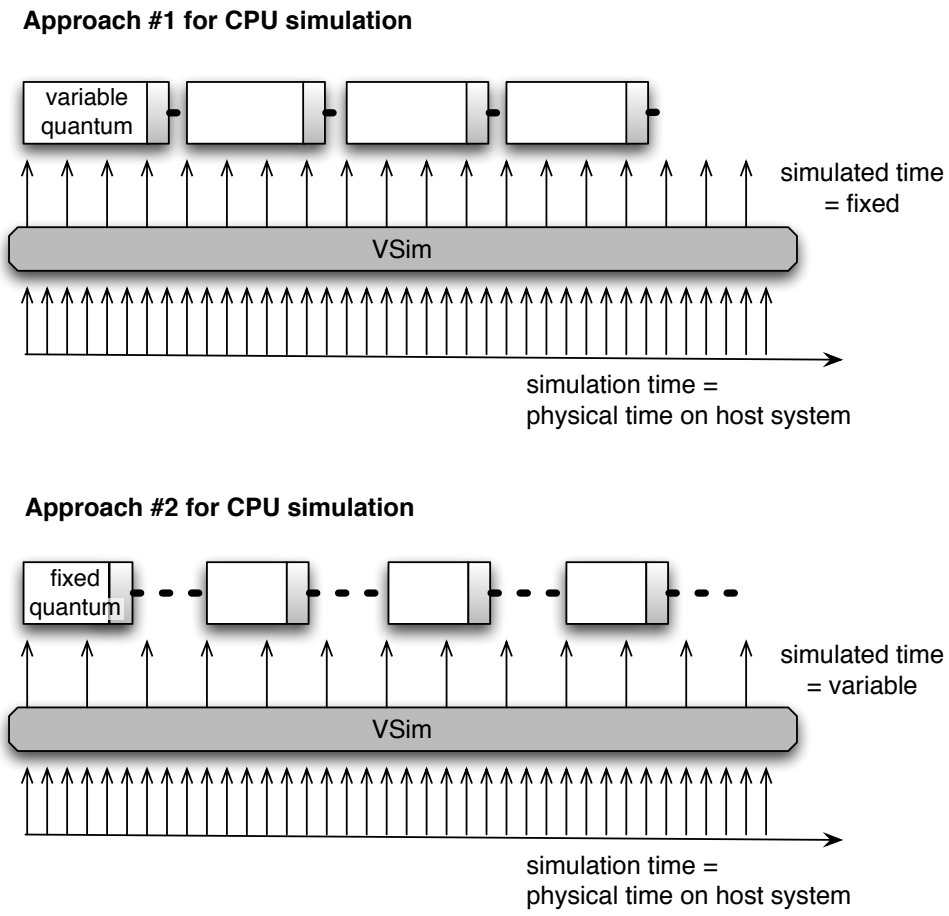

Figure 3.3: Two approaches for CPU simulation.

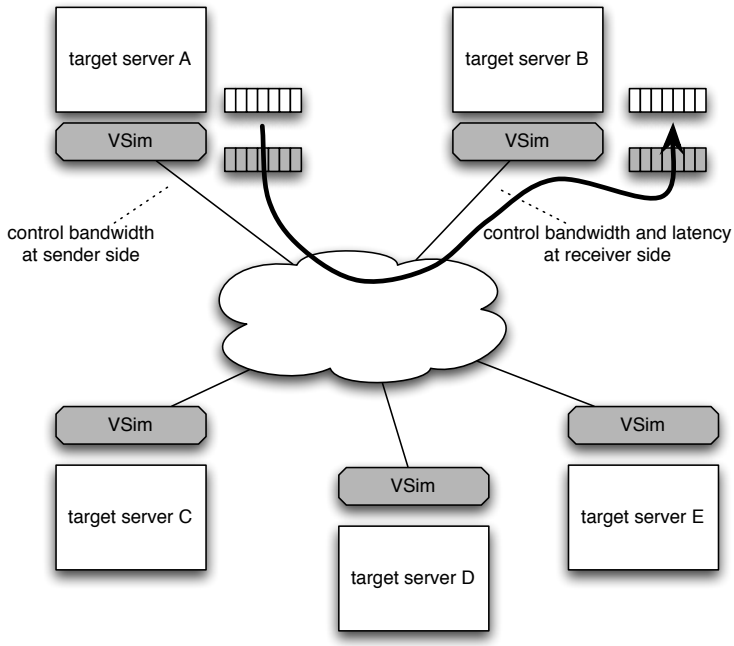

Figure 3.4: Network simulation in VSim.

simulated servers, and a mechanism would need to be put in place to synchronize time among the simulated servers. Approach #1 does not need such a synchronization mechanism.

Our current implementation of VSim requires its users to specify the speed of the target system relative to the host system. For example, if one wants to simulate a particular workload on a target server that achieves twice the performance of the host server, VSim will execute two units of work per simulation quantum, following approach #1. The relative speed between the target system and host system is workload-dependent. For example, the relative performance difference between a high-end processor and a low-end processor is likely to be different (i.e., smaller) for a memory-intensive workload compared to a compute-intensive workload.

One potential avenue for future work could be to leverage performance counters on the host and build performance models that estimate target performance based on host performance counter values. This would eliminate the need for offline profiling to determine relative host/guest performance, and it would enable accounting for an application's time-varying execution behavior.

3.2.3 Network simulation

In order to be able to simulate a target network configuration on a host network, one needs the ability to control network bandwidth and latency. VSim does this by intercepting all network I/O and by manipulating bandwidth and latency. Figure 3.4 illustrates this. At the sender side, VSim controls the outgoing network bandwidth. This is done by keeping a buffer in VSim that holds all the outgoing network packets and sends out the packets at the bandwidth of the target network link. At the receiver side, VSim accepts all the incoming packets and holds them temporarily in a buffer. VSim delivers the packets to the target software stack at the rate determined by the target NIC bandwidth. When to deliver the packets to the target software stack is determined by the network latency. This is done through a table lookup based on the packet's sender IP address. Adjusting the sender-to-receiver network latency enables modeling different network topologies, e.g., different latencies need to be set for mesh, ring and tree networks. Other network topologies and policies can be modeled as well, including QoS-aware policies with specific bandwidth and latency properties between specific nodes in the network, and/or bandwidth and latency properties that depend on packet priority.

3.2.4 Disk simulation

Figure 3.5 illustrates how we simulate disk I/O in VSim. On a target system, the CPU sends the disk request. A DMA transfer is then initiated to copy the data between disk and main memory, and the CPU resumes doing useful work. In the simulated system, VSim intercepts the disk request, the DMA transfer is initiated and the CPU is stopped (i.e., VSim no longer schedules units of work and simulated time is stopped for the target CPU). When the DMA transfer is completed, the disk sends an interrupt to the CPU, which is intercepted by VSim. The interrupt is held back by VSim. VSim reschedules the target CPU which can then perform useful work and advance simulated time. The amount of work done by the target CPU (also the time during which the interrupt is held back) is determined by the latency of the (simulated) disk operation. The disk operation latency is determined by whether the disk operation is a read or write, the size of the disk request (how many blocks need to be written or read), the offset on the disk, and the disk status. VSim signals to the target CPU when the disk operation has been completed, and delivers the interrupt.

The reason for stopping the CPU while sending the disk request to the disk on the host, is to be able to simulate an SSD target disk on an HDD host disk. An SSD disk features shorter latencies, hence, stopping the CPU while the host disk fetches the data and rescheduling the CPU for a period

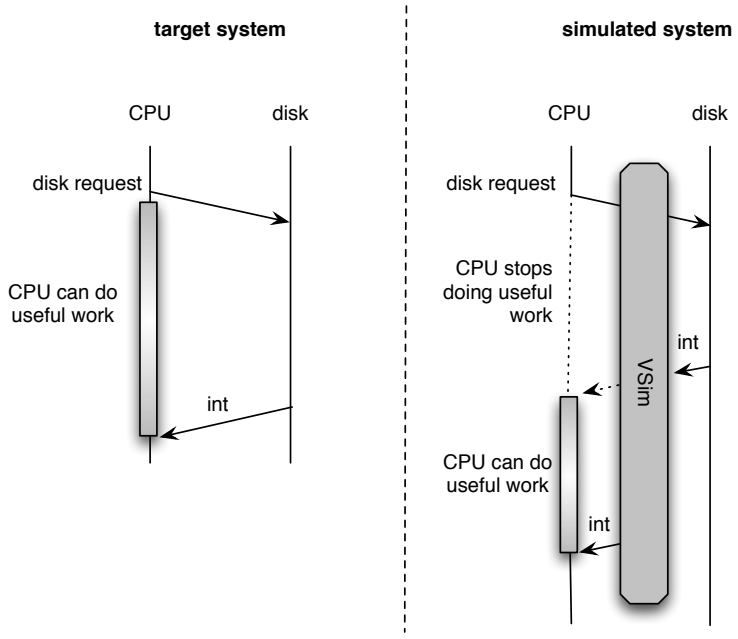

Figure 3.5: Disk simulation in VSim.

of time equal to the access latency of the target disk is the only way one can simulate a fast disk on a slow one.

Stopping the CPU upon a disk I/O request has its implications for keeping the various simulated servers synchronized, i.e., simulated time is stopped for CPUs that perform disk I/O while simulated time advances for the other CPUs. Without appropriate counteractions, this would cause simulated time to diverge across the simulated servers. VSim accounts for this by keeping track of how long a CPU is stopped while performing the physical disk access on the host server. At the end of the simulation quantum, the CPUs that did not perform disk I/O are stopped to allow for the CPUs that did perform disk I/O to catch up.

3.2.5 Synchronizing across host servers

So far, we were concerned with simulating multiple target servers on a single host server, and we already dealt with the issue of synchronizing simulated time across the simulated target servers in this setting. However, eventually, the goal is to simulate a multitude of target servers on a number of host servers. This brings in the difficulty of synchronizing simulated time across the various host servers. This is the classical simulated time synchronization problem in parallel discrete event simulation (PDES) [32].

There exist a number of approaches for how to deal with this synchronization problem in distributed simulation. One approach is to synchronize at each simulation quantum [33], i.e., all the simulated servers (on the various hosts) need to reach the end of the simulation quantum before simulation is started for the next quantum. While this would yield high accuracy, it is likely to have an impact on simulation speed which is bounded by the slowest simulated server. At the opposite side of the spectrum, lax synchronization does not synchronize between the simulated servers and lets all simulated servers run freely. Between these two opposite ends of the spectrum one can imagine different policies [26,34,35], e.g., synchronize every n simulation quanta, or track the delta in simulated time between the various simulated servers and hold simulated servers for which this delta exceeds a specified threshold. VSim currently implements lax synchronization, but other synchronization mechanisms that balance simulation accuracy and speed can be implemented as well.

3.2.6 Strengths

VSim has a number of strengths.

- VSim can simulate different multi-server architectures by varying CPU, network and disk performance. The architecture parameters that can be varied are the number of CPUs, CPU performance, network bandwidth and latency, and disk latency. VSim can also be used to model heterogeneous system architectures with different types of CPUs, disks, and network topologies and configurations.
- VSim can simulate entire software stacks. VSim is implemented in a system virtual machine — either a hosted virtual machine or a hypervisor — hence, it runs an unmodified operating system and a complete application stack on top of it, including virtual machines, middleware, application servers, etc.
- VSim can simulate a large target system on a small host system, i.e., VSim simulates multiple target servers on a single host server through multiplexing.
- VSim exploits multi-threaded parallelism in the host server: simulating multiple target servers is parallelized on the host server by running each core of the target servers on the different cores and/or SMT hardware threads on the host.
- VSim is a scalable simulation approach and allows for simulating multi-server setups on a smaller number of host servers. In other words, VSim is a distributed simulation approach and time is synchronized across the various simulated servers.

- VSim runs at near native hardware speed. Simulation speed is limited by the number of target servers one wants to simulate per host server and the virtualization overhead. In our setup, the simulation slowdown of VSim is limited to one order of magnitude (10×) relative to native hardware execution while being able to simulate up to 5 target machines per host machine.

3.2.7 Limitations

In spite of the important strengths mentioned above, VSim also has some limitations.

- VSim's scalability is limited by the amount of physical memory in the simulation host. VSim needs to keep track of the memory state for all the target servers that it simulates per host server. This limitation can be overcome by adding more main memory to the host. Or, in addition, memory page sharing among the guests (the simulated servers) can lower memory pressure on the host server [36].
- For a given workload, the performance of the guest server is modeled as a fixed factor of the host platform's performance. Determining relative guest/host performance is done offline. However, as mentioned before, one could leverage performance counters on the host to predict target performance in an online fashion, which would also enable modeling time-varying workload behavior; this is an interesting avenue for future work.
- Processor and memory subsystem performance is modeled as a single performance factor. By using hardware performance counters and an online performance model, as mentioned above, one could more accurately determine the impact of CPU and memory performance; again, we leave this for future work.
- VSim does not provide enough detail for performing detailed microarchitecture simulations. Instead, VSim's key feature is to simulate and provide insight in the scaling behavior of multi-server workloads running on (potentially heterogeneous) multi-server setups.

3.3 Experimental Setup

3.3.1 VSim implementation and configuration

We implemented a VSim prototype in VirtualBox v3.1.2. VirtualBox is a hosted system virtual machine — a hosted VM runs on top of an operating

	Intel Atom	AMD Opteron 2212	AMD Operton 2350
Core type	in-order	out-of-order	out-of-order
# cores	2	2	4
Frequency	1.6 Ghz	2 Ghz	2 Ghz
L1 cache	56 KB private	128 KB private	128 KB private
L2 cache	512 KB private	1 MB private	512 KB private
L3 cache	—	—	2 MB shared

Table 3.1: CPUs considered in the evaluation.

HDD	Western Digital Caviar Blue 500 GB, SATA 3 Gb/s	
		16 MB cache, 7200 rpm
	IOzone:	sequential read 29.6 MB/s, random read 0.6 MB/s
		sequential write 27.6 MB/s, random write 0.3 MB/s
SSD	Intel X25-M 80 GB, SATA 3 Gb/s	
	IOzone:	sequential read 36.6 MB/s, random read 16.7 MB/s
		sequential write 30 MB/s, random write 28.5 MB/s

Table 3.2: The SSD and HDD disks considered in the evaluation along with their properties according to IOzone.

system in contrast to a hypervisor or virtual machine monitor which runs on bare metal. We set the simulation window to 10 ms and the simulation quantum to 100 ms in all of our experiments. We experimentally evaluated different values for the simulation window and quantum, and we found the above values to be effective. This setting yields a 10× simulation slowdown compared to native hardware execution in all of our experiments.

3.3.2 Hardware platforms

We consider three CPUs: Intel Atom 330, AMD Opteron 2212 and AMD Opteron 2350. These machines have vastly different properties as shown in Table 3.1. The Intel Atom processor is a dual-core multi-threaded in-order processor [37] whereas the AMD Opteron processors are multicore out-of-order processors. The AMD Opteron machines both implement the K10 microarchitecture [12] but their multicore architecture differs: the 2212 features two cores and private 1 MB L2 caches, and the 2350 features 4 cores with private 512 KB L2 caches and a shared 2 MB L3 cache.

We consider two disk types in our evaluation, a hard disk drive (HDD) and a solid state disk (SSD), see Table 3.2. The HDD and SSD differ substantially in terms of their properties. We quantify the various disk parameters, such as sequential read/write bandwidth and random read/write bandwidth, using IOzone which is a filesystem I/O benchmark [38].

Benchmark	Suite	Description
blackscholes	PARSEC	Option pricing with Black-Scholes PDE
blastp	BioPerf	Identification of similar protein sequences in a database
bodytrack	PARSEC	Body tracking of a person
ce	BioPerf	Finds structural similarities between pairs of proteins
ferret	PARSEC	Content similarity search server
freqmine	PARSEC	Frequent itemset mining
h264dec	MediaBench II	H.264 video decoding
h264enc	MediaBench II	H.264 video encoding
raytrace	PARSEC	Real-time raytracing
specjbb2005	SPECjbb	Evaluates Java server performance.
streamcluster	PARSEC	Solves online clustering problem
swaptions	PARSEC	Pricing of a portfolio of swaptions

Table 3.3: CPU benchmarks used in this study.

3.3.3 CPU benchmarks

Table 3.3 lists the CPU benchmarks used in this study. They are taken from a variety of sources such as PARSEC [15], BioPerf [16], MediaBench II [17] and SPECjbb2005; the PARSEC benchmarks are multi-threaded and model Recognition, Mining and Synthesis (RMS) workloads.

The target servers (as well as the simulated servers) run the Ubuntu 9.04 server operating system. The Java virtual machine involved in some of our workloads is the Sun JRE6.

3.4 Evaluation

We now validate and evaluate VSim. This is done in a number of steps. We first validate VSim against real hardware and we consider the various subcomponents in isolation: CPU, network and disk. Subsequently, we put it together and validate VSim for simulating a client-server setup including CPU, network and disk. Finally, we demonstrate the utility of VSim for exploring the system architecture of a multi-tier Web 2.0 server application along with a 25-server Hadoop workload setup. For all of our experiments we do 20 runs (both on hardware and within the VSim simulator), and we report 99% confidence intervals.

3.4.1 CPU validation

We first focus on validating the CPU model in VSim. We consider two target servers, the Intel Atom machine and the AMD Opteron 2350 server; the host server in all of our experiments is the AMD Opteron 2212 with HDD.

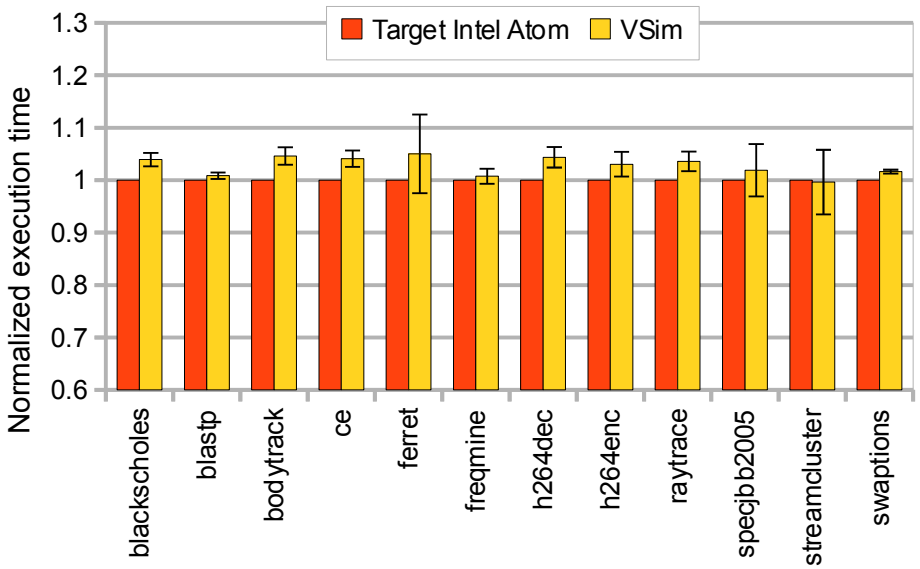

Figure 3.6: Validation of the CPU model of the Intel Atom target on the AMD Opteron 2212 host.

Validation

In these experiments we first run the benchmarks on the target processors and the host processor, and we determine the performance of the target processor relative to the host processor. All benchmarks run a single thread for now; we consider multi-threaded runs later. We clustered the benchmarks into three groups according to their relative performance. There are three groups for the Intel Atom target: 60% performance relative to the AMD Opteron 2212 host (or 40% worse performance on the Intel Atom compared to the AMD Opteron 2212; this cluster includes ferret), 30% relative performance (includes blastp and swaptions), and 40% relative performance (includes all the other benchmarks); similarly, there are three groups for the AMD Opteron 2350 target: 170% relative performance (70% better performance; includes ferret and streamcluster), 130% relative performance (includes ce and SPECjbb), and 100% (same performance; includes all the other benchmarks)[1]. We refer to these clusters as the 'target Intel Atom' and the 'target AMD Opteron 2350', respectively, as this is the target performance VSim should model. We then run each of the benchmarks in VSim on the host server, and we set the performance target according to the above performance targets, e.g., the ferret benchmark is run at a 60% and

[1]Classifying benchmarks into three clusters introduces an average error of 7.6% and a maximum error of 26.8% for blastp on the Intel Atom, and an average error of 5.3% and a maximum error of 11.7% on the AMD Opteron 2350. These errors are a result of the clustering, and not VSim.

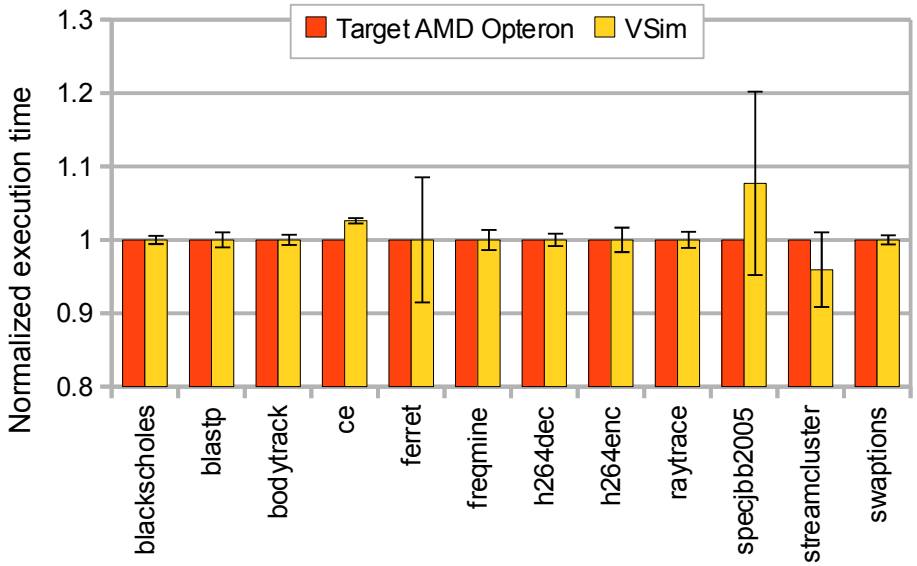

Figure 3.7: Validation of the CPU model of AMD Opteron 2350 target on the AMD Opteron 2212 host.

170% performance target when simulating the Intel Atom and the AMD Opteron 2350, respectively. We then report simulated time and compare against the target performance numbers.

The results of this validation experiment are shown in Figures 3.6 and 3.7 for the Intel Atom and AMD Opteron 2350, respectively. These graphs report normalized execution time for the two target platforms, and the simulated execution time by VSim. The ideal target execution time falls within the confidence intervals of the simulated execution times for all benchmarks, with an average error of 2.85% and 1.20% for the Intel Atom and AMD Opteron, respectively. These results demonstrate VSim's ability to accurately model CPU performance.

Scalability

The above experiments simulated a single target on a single host. We now evaluate VSim's scalability in terms of how many targets VSim can simulate on a single host with good accuracy. Figure 3.8 shows the average execution time across all benchmarks when simulating multiple targets normalized to simulating a single target, i.e., we run multiple copies of the same target on a single host and compute the average execution time reported by VSim across the targets; we then normalize against the execution time for a single target, and we then report the average across all of the benchmarks. The error remains small (less than 5%) for an increasing

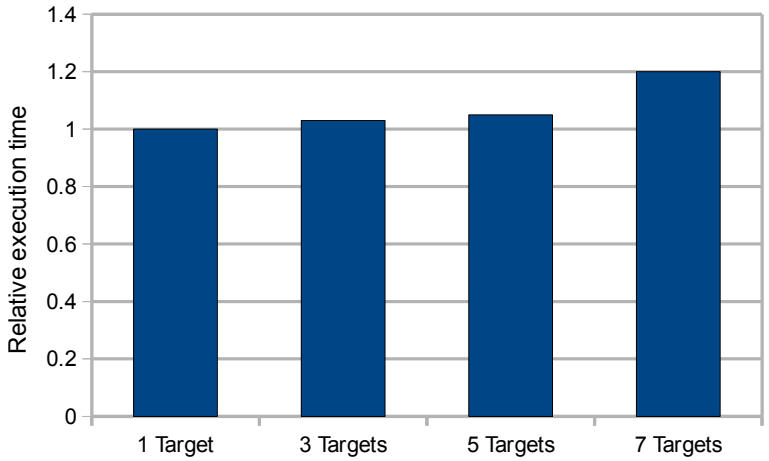

Figure 3.8: Evaluating VSim scalability in terms of simulating multiple targets per host.

number of targets, up to 5 targets. The error increases to almost 20% for 7 targets. The reason for this increase is that VirtualBox (in which VSim is currently implemented) is a hosted virtual machine. This implies that VSim is not in control when it comes to scheduling the target server simulations; instead, the underlying operating system is. As a result, the operating system may dynamically change the order of the target simulations, and target simulations may need to wait before being rescheduled. The waiting time however is viewed by VSim as sleep time. Hence, the actual simulated execution time may diverge from what is perceived by VSim, and hence simulation errors are introduced. If VSim were in control of scheduling its target simulations — e.g., if VSim were implemented in a hypervisor — it is likely to expect that VSim will scale further beyond 5 targets per host. The maximum number of simulated targets is 10 given the 10 ms simulation window versus 100 ms simulation quantum settings, and can only be achieved if VSim's virtualization overhead is very small.

Multi-threaded workloads

Figure 3.9 shows results for the multi-threaded benchmarks in our benchmark suite, with each benchmark running four threads. We model the AMD Opteron 2350 as the target on the AMD Opteron 2212 host. We consider two versions of VSim here. The first version, called 'VSim' in Figure 3.9, does not synchronize the scheduling of the threads, i.e., this implies that threads may be scheduled in subsequent simulation quanta. The second version on the other hand, called 'VSim + synchronization' in Figure 3.9, synchronizes thread scheduling: it co-schedules the threads on sep-

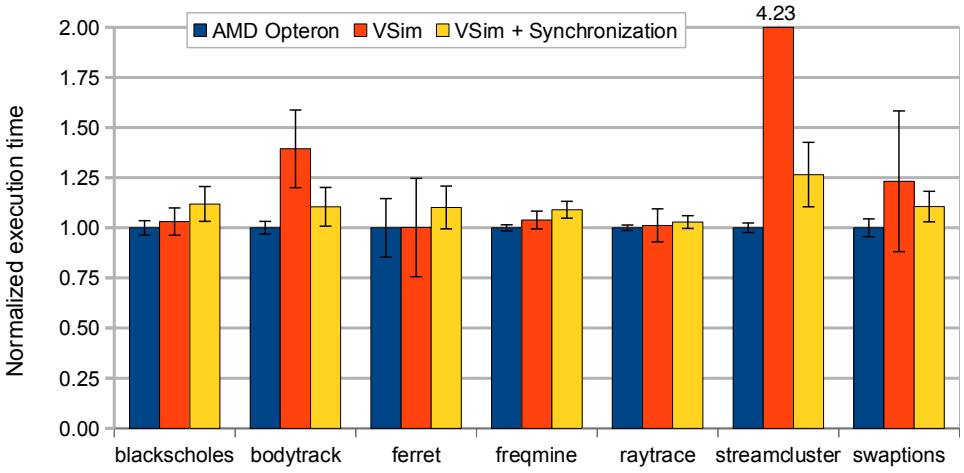

Figure 3.9: CPU validation for multi-threaded workloads.

arate cores in the same simulation quantum. We observe good accuracy for all benchmarks when co-scheduling the threads. When not synchronizing thread scheduling, we observe large errors for one benchmark, streamcluster, because of its lock-intensiveness: timing-sensitive behavior at a granularity smaller than the simulation quantum is unlikely to be accurately modeled because a lock held by a thread upon the end of a simulation window will be held until the next simulation quantum, which prevents other contending threads from making progress. This problem is overcome by co-scheduling lock-intensive threads, see the 'VSim + synchronization' bars in Figure 3.9.

3.4.2 Network validation

We now validate the network simulation approach in VSim. We consider a 1 Gbit host network and model 1 Gbit and 100 Mbit target networks at a simulation slowdown of 10×. The workload considered here is ftp which transfers an 800 MB file between two servers. We report throughput in Megabytes per second and validate the simulation results against real hardware, see Figure 3.10. The throughput numbers reported by VSim very closely match the throughput numbers obtained on real hardware for both the 100 Mbit and 1 Gbit target networks: the real numbers fall within the confidence interval obtained through simulation. We obtain an average error of 4.4% and a maximum error of 15% for the 1 Gbit network, and an average error of 3.6% and a maximum error of 6% for the 100 Mbit network.

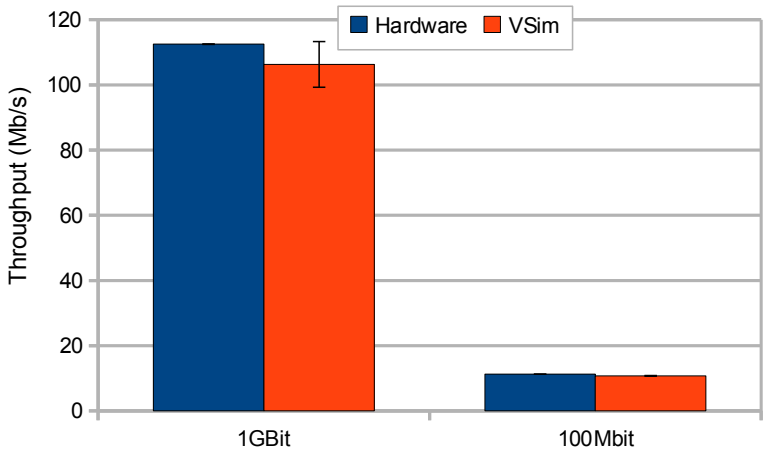

Figure 3.10: Network validation.

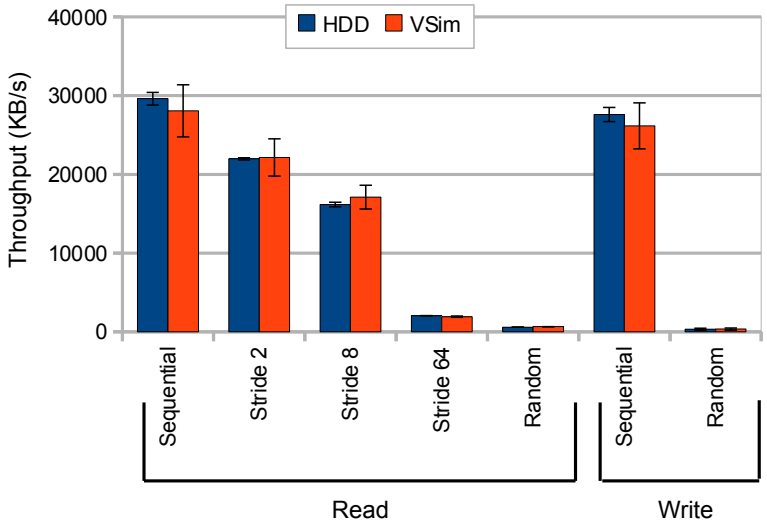

Figure 3.11: Disk HDD validation.

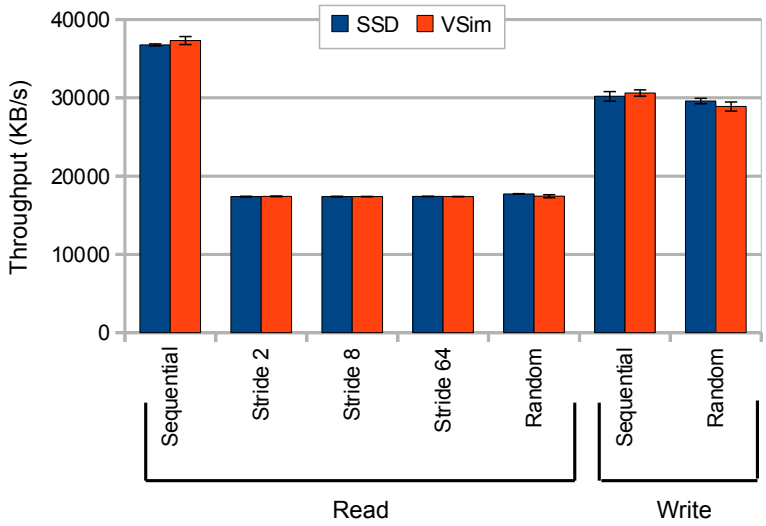

Figure 3.12: Disk SSD validation.

3.4.3 Disk validation

Figures 3.11 and 3.12 validate the disk models in VSim for the HDD and SSD, respectively, using the IOzone filesystem benchmark[2]; the HDD is the host. We make a distinction between reading and writing the disk, and we consider different disk access patterns: sequential, strided and random. Sequential access means accessing subsequent 4 KB blocks; strided access of 2 means accessing every other 4 KB block on disk; etc. VSim's accuracy is good: the real hardware measurements always fall within the confidence intervals of the simulation. VSim is able to accurately model the disk latency difference between HDD and SSD. We report an average error of 3% and maximum error of 6% for SSD, and an average error of 5% and maximum error of 10.7% for HDD. SSD achieves higher disk bandwidth for sequential reads, big strided reads (64 blocks), as well as random reads and writes; HDD achieves a higher throughput for 2-strided reads compared to SSD. VSim tracks these differences in disk throughput accurately compared to real hardware.

3.4.4 Lucene indexing benchmark

Now that we have validated all of the VSim subcomponents, we are ready to put everything together and evaluate VSim's accuracy while considering CPU, network and disk. We consider a text search engine written in Java, called Lucene, and we consider two experiments. In the first experiment,

[2]http://www.iozone.org/

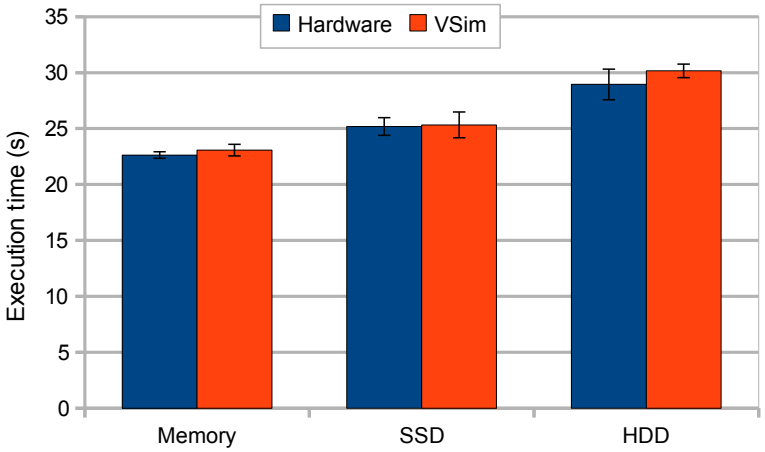

Figure 3.13: Lucene index building on 10,000 Wikipedia documents held in memory, SSD and HDD.

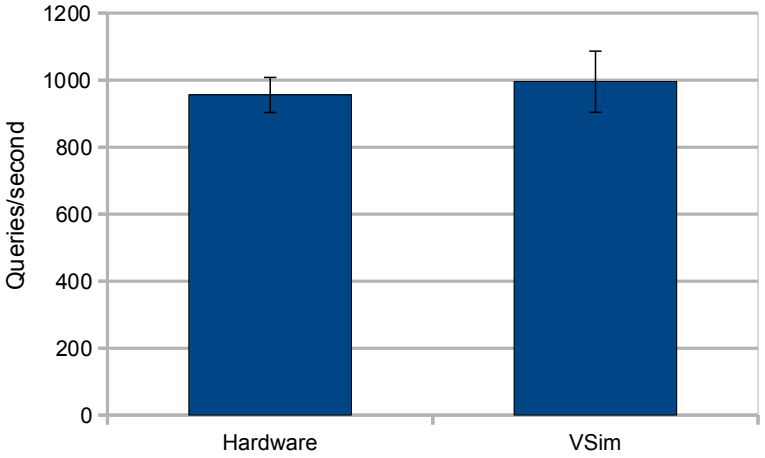

Figure 3.14: A client-server setup with a client modeling multiple concurrent users querying the Lucene index stored on the server.

Lucene builds up an index for 10,000 Wikipedia documents and we compare three scenarios: the Wikipedia documents are held in (i) main memory, (ii) SSD and (iii) HDD. Figure 3.13 compares the simulated execution time using VSim run on the AMD Opteron 2212 (the host, with HDD) — note the entire system, including CPU, network and disk, is simulated — against the execution time on the AMD Opteron 2350 (the target). We report an average error of 2.2% and a maximum error of 4.2%. The confidence intervals for the simulation and real hardware execution times overlap.

In the second experiment we consider a client-server setup. The client simulates a number of concurrent users sending queries to the Lucene index stored on the server. We use siege as a stress test on the client side; the clients send requests to the server with zero think time. The client is run on the AMD Opteron 2212 and the server is run on the AMD Opteron 2350; further, the Lucene index is stored on HDD, and the client and server are connected through a 1 Gbit network. VSim simulates the client and server machines on a single host machine (the AMD Opteron 2212). Figure 3.14 compares VSim against the hardware experiment. Again, VSim is accurate compared to real hardware as the confidence intervals overlap with an average error of 4.2% (maximum error of 14%).

3.4.5 Case study #1: Olio Web 2.0

As a case study to illustrate VSim's utility for driving multi-server setup design decisions and optimizations, we consider the Olio benchmark — which is also used in Cloudstone [39], a benchmark developed for evaluating Web 2.0 and cloud computing performance. Olio is a realistic Web 2.0 social-events application: a client (rain [40], a Markov-chain based workload generator) sends requests to the Web server which then interfaces with a file server and a database server. We assume that each of the servers (Web server, file server and database server) runs on a separate physical server. We explore different trade-offs in performance versus cost while changing on which CPU node each server is run. The total cost of ownership (TCO), or cost for short, includes hardware purchasing cost and energy cost (both empowering and cooling the servers) assuming a 3-year depreciation. We consider three configurations, from left to right in Figure 3.15: (i) all servers run on Intel Atom 330 nodes, (ii) the Web server is run on AMD Opteron 2350 and the file and database servers are run on Intel Atom nodes, and (iii) all servers run on AMD Opteron 2350 nodes. We run the various target servers on a single host (the AMD Opteron 2212) at a 10× simulation slowdown. The interesting observation is that the heterogeneous configuration (middle configuration) — Web server run on AMD Opteron 2350, and the file and database servers run on Intel Atom 330 — yields the best performance-cost trade-off for this particular workload. Its performance is

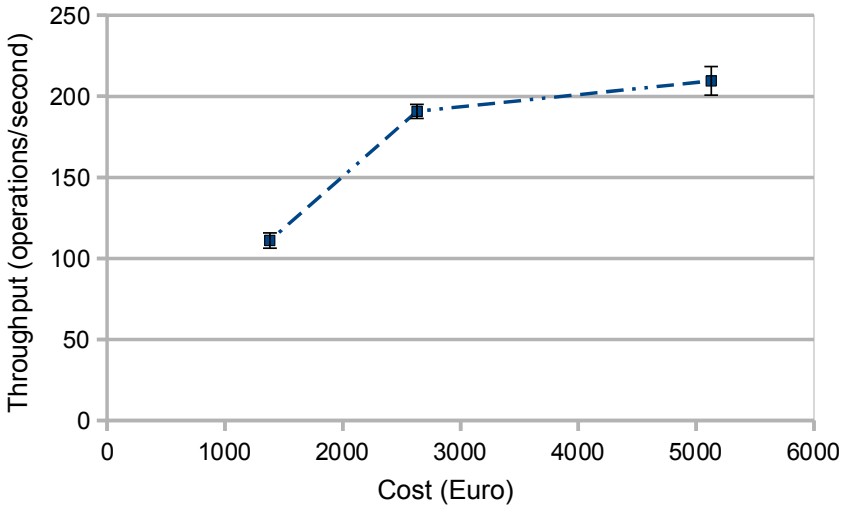

Figure 3.15: Comparing three datacenter configurations for the Olio Web 2.0 benchmark in terms of performance (vertical axis) and cost (horizontal axis): (leftmost point) all servers run on Intel Atom 330 nodes; (middle point) the Web server is run on AMD Opteron 2350 and the file and database servers run on Intel Atom 330; (rightmost point) all servers run on AMD Opteron 2350 nodes.

nearly as good as running all three servers on AMD Opteron 2350, yet its cost is significantly lower and its performance is substantially better than running all servers on Intel Atom. The reason is that the Web server caches file and database requests using a memcached service, i.e., a memcached service runs on the Web server and handles recently accessed files and database records, and thereby reduces the load on the file and database servers. As a result, less powerful server nodes for the file and database servers achieve comparable overall system performance, yet their cost (and energy consumption) is substantially lower.

3.4.6 Case study #2: 25-server Hadoop workload

Our second case study involves a Hadoop workload with a distributed (MapReduce-style) version of word count on 100,000 Wikipedia documents. We use Apache Hadoop 0.20, the OpenJDK 6 JVM and the Ubuntu 10.04 server on the software side. Our target platform consists of 25 AMD Opteron 2212 servers with HDD interconnected through 1 Gbit ethernet. We simulate all target servers on five host servers at a slowdown of one order of magnitude. Figure 3.16 shows the throughput as a function of the number of target servers. The graph shows two curves: the simulated curve with up to 25 target servers along with the real hardware curve up

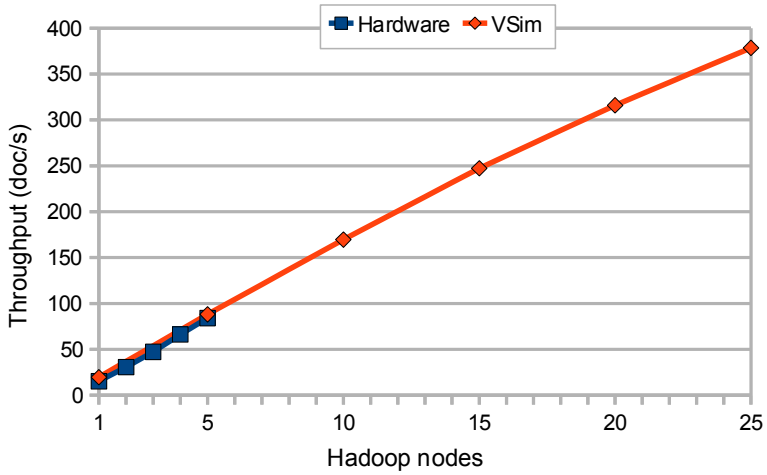

Figure 3.16: Simulating up to 25 target servers running the Hadoop workload on 5 host servers.

to 5 servers. VSim achieves the same throughput as the real hardware — we cannot validate beyond 5 target servers because we do not have access to more than 5 target servers. We observe almost linear scaling, although we observe a slightly sublinear scaling beyond 20 target servers. The key message is that VSim can be used for studying scale-out behavior.

3.5 Related Work

The simulation approach most closely related to VSim probably is COTSon (also used and improved upon in the previous chapter). COTSon [8] is an open-source simulation framework developed by HP Labs. COTSon targets cluster-level systems consisting of multiple multicore processor nodes connected through a network, i.e., it targets both scale-up (i.e., multicore and manycore processor simulation) as well as scale-out (i.e., simulation of a multinode cluster). COTSon uses the AMD SimNow full-system simulator to functionally simulate each node in the cluster. Each COTSon node further consists of timing models for the disks, network card interface and the CPU (i.e., processor and memory). The various COTSon nodes are interconnected through a network mediator. There is at least one key difference between VSim and COTSon. VSim models target performance by moderating the amount of work done per simulation quantum for each of the simulated targets. COTSon on the other hand uses a performance model that is fed by a dynamic trace of instructions which is analyzed and based on which performance is estimated. These differences lead to different trade-offs in the simulation space. VSim is faster (10× slowdown

versus 50× for COTSon) and consumes much less memory (20 MB memory overhead for VSim versus 2 GB for COTSon), hence, VSim can simulate both more targets per host and larger systems at higher speed compared to COTSon — VSim is more scalable. A limitation of VSim is that it is tied to the host server's instruction-set architecture (ISA); this is not necessarily the case for the COTSon approach which leverages functional simulation rather than virtualization.

SimOS [41] was the first simulator achieving the high execution speeds required to simulate a complete system including the operating system. Binary translation was employed to achieve this high execution speed. SimOS includes CPU, disk an network simulation making it suitable for simulating networked systems. SimOS formed the foundation for the VMware hypervisor; hardware vendors included special instructions to speed up system virtualization. VSim uses hardware assistance in combination with time dilation to achieve high simulation speeds, and it enables simulating multiple guests on a single host.

Open Cirrus [39] is an open cloud-computing research testbed that was initiated by a collaborative group of researchers in both industry and academia. Open Cirrus' primary goal is to provide a distributed set of federated datacenters as a testbed for system-level cloud computing research: it provides open-source software stacks and APIs, it enables systems-level research, it provides experimental data sets and it allows for studying application development for cloud computing.

Gupta et al. [42] propose time dilation for network simulation. Time dilation provides the illusion to an operating system and its applications that time is passing at a rate different than physical time. They implement time dilation in the Xen virtual machine by filtering delivered timer interrupts. Gupta et al. use time dilation for simulating network devices only. VSim in contrast models complete computer systems, including CPU, network and disk activity through time dilation. Moreover, VSim enables running multiple target servers per host server — this was not explored in the Gupta et al. paper — which is required for simulating scale-out scenarios.

An alternative approach to building simulators and testbeds is to build high abstraction models. For example, Ranganathan and Leech [43] use utilization traces from real deployments in conjunction with high-level models that correlate resource utilization to power and performance. Weisner and Wenish [44] model datacenter workload behavior using queuing models.

3.6 Conclusion

Simulating multi-server systems is challenging. It requires the ability of running complete software stacks, while modeling CPU, disk and network activity. In addition, it needs good simulation speed and accuracy, while being scalable. This chapter presented VSim, a novel simulation paradigm which offers a unique trade-off in speed versus accuracy versus scalability while being able to model complete systems — a promising approach for simulating systems at scale. VSim leverages virtualization technology to run multiple target servers as guests on a host, and manipulate CPU, disk and network performance as observed by the guests through time dilation such that the software stacks are given the illusion to run on the target system. The implementation of VSim in VirtualBox and the evaluation presented in this chapter illustrate its accuracy: 2.0%, 4.4% and 4.9% average error for modeling CPU, disk and network performance, respectively; complete workloads (Lucene and Olio) involving CPU, disk and network activity are shown to be accurately modeled in VSim (average error of 3.2%). These results are obtained at a simulation slowdown of one order of magnitude only compared to native hardware speed, and our current implementation can simulate up to five target servers per host. We reported on two case studies illustrating how VSim can be used for making design trade-offs and for exploring workload scaling behavior.

Chapter 4

Trends in Computer System Energy Proportionality

Having studied and developed simulation methodologies in the previous two chapters, we now move to real hardware experiments and measurements with a specific focus on energy-efficiency. This chapter analyzes trends in energy-proportionality of contemporary servers. Using power/performance numbers from a broad set of commercial machines, we analyze how energy-proportionality has evolved over the past three years. We evaluate to what extent SPECpower quantifies energy-proportionality, and we study how much total energy can be saved by making servers even more energy-proportional.

4.1 Introduction

Energy efficiency has emerged as a major technology driver in servers and datacenters today, as it impacts capital cost as well as operating expenses for powering and cooling the servers. Energy-related costs have become an important component in the total cost of ownership (TCO) of this class of systems. In fact, Barroso and Hölzle [1] report that server capital cost dominates overall TCO in a classical datacenter, with 69% of the monthly cost being related to server purchase and maintenance. In contrast, a contemporary datacenter with commodity-based lower-cost servers and/or higher power prices is very different: the cost of all infrastructure and power to host the datacenter is more than twice the purchase and maintenance cost. They conclude that, with electricity and construction costs trending up, datacenter facility costs (which are proportional to power consumption) will become an increasingly larger part of the total cost. In other words, increasingly, the total cost of a datacenter will be primarily a function of the power it consumes, and the purchase and maintenance cost will matter increasingly less. Besides cost considerations, improving energy efficiency

is key to reduce carbon dioxide emissions by the IT industry, which should eventually lead to 'greener' IT.

Improving a server's energy efficiency is non-trivial and faces many challenges. Peak power consumption, for one, affects capital cost for the power distribution and supply units and cooling infrastructure. Also, high power consumption leads to increased power density and high temperature which affects cooling costs and which may affect hardware reliability and availability. Total energy consumption affects the operating expense for powering the servers. Further, Fan et al. [45] report that the most common operating mode for servers is in the 10 to 50 percent range, i.e., servers in large datacenters are rarely completely idle and seldom operate at or near maximum utilization; instead, servers operate in the 10 to 50 percent utilization range for the majority of the time. This insight motivated Barroso and Hölzle [46] to make the case for energy-proportional servers, or servers that consume energy proportional to their utilization level or load. The key insight is that servers should not only be optimized for reducing peak power consumption, but they should also be optimized for their most common operating points at lower utilization levels.

In December 2007, SPEC released SPECpower_ssj2008 [10], an industry-standard benchmark that measures power and performance for servers. SPECpower measures performance and power at different utilization levels from which it computes an overall metric. SPECpower thus includes some notion of energy-proportionality, however, it does not explicitly quantify energy-proportionality.

In this chapter, we propose the Energy Proportionality (EP) metric to quantify a server's energy-proportionality. Using the EP metric on published power and performance data, we evaluate how energy-proportionality has evolved over time, we discuss how the EP metric relates to the established SPECpower metric, and we quantify how much total energy can be reduced by further improving servers' energy-proportionality.

4.2 Energy-proportionality of contemporary servers

The concept of energy-proportionality can be easily explained through an analogy. A car that is parked does not consume fuel. On the other hand, when the engine is running, it does burn fuel, and it consumes even more fuel when the car accelerates. However, when waiting at the traffic lights, the engine also burns fuel, even if the car is not driving. The latter is an example of a case where the car is not energy-proportional: it consumes energy (i.e., fuel) although it does not make forward progress. Similarly, a computer system consumes energy, i.e., the computer system is powered on, although it does not do any useful work.

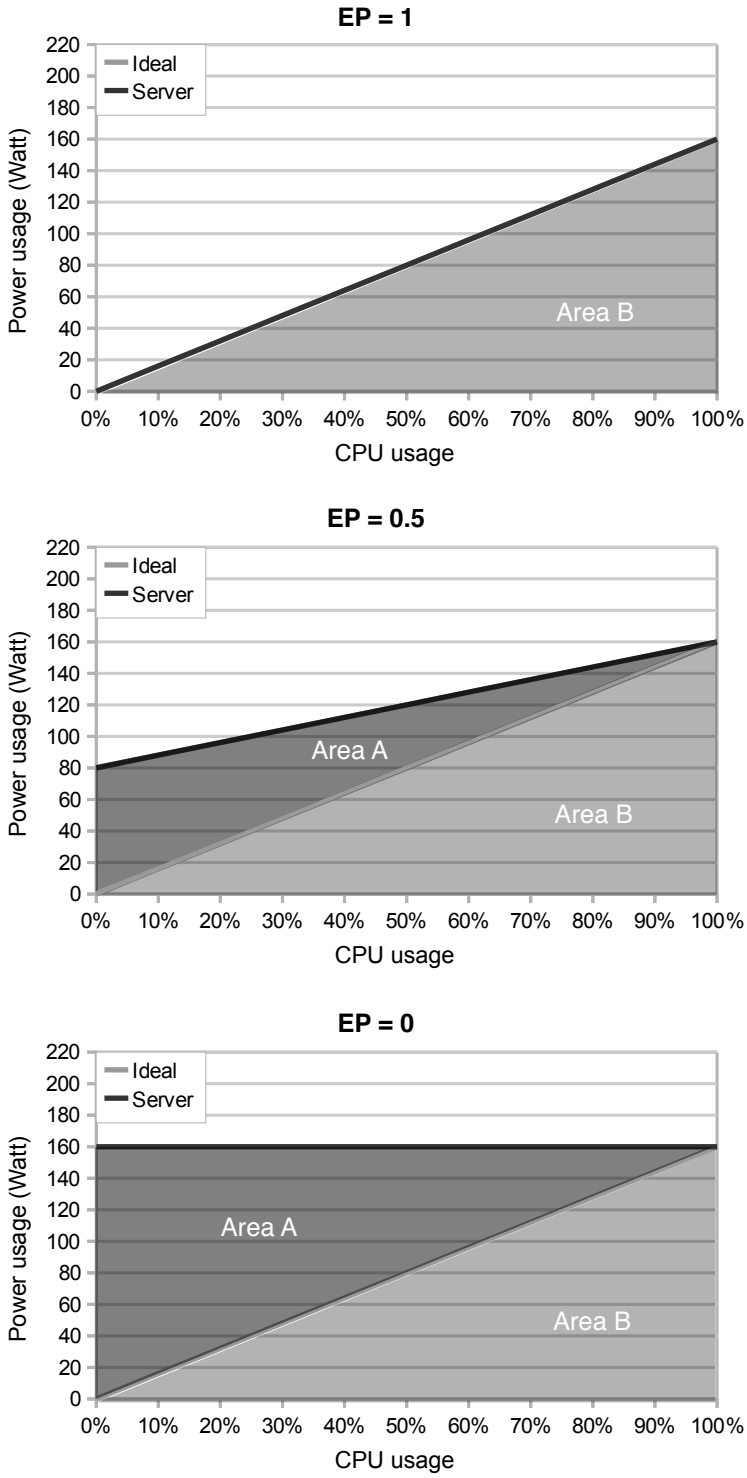

Figure 4.1: Energy-Proportionality (EP) is defined as one minus Area A divided by Area B. Top graph shows perfect energy-proportional system ($EP = 1$); bottom graph shows a non-energy-proportional system ($EP = 0$); and the middle graph shows a 50 percent energy-proportional system ($EP = 0.5$).

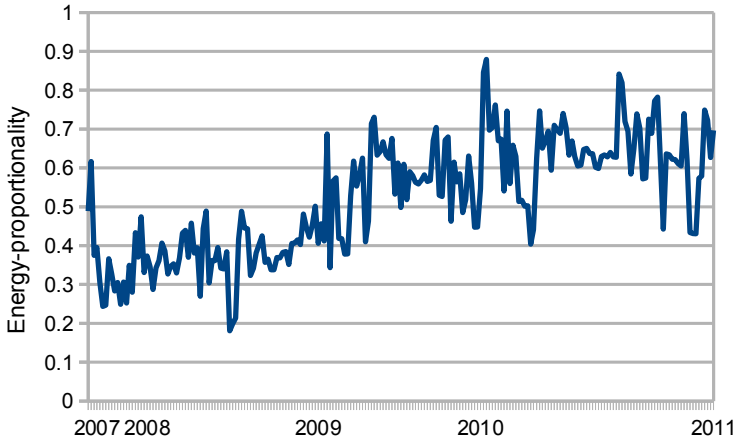

Figure 4.2: Energy-Proportionality (EP) over time.

Ideally, an energy-proportional computer system would consume zero power when completely idle, and would consume power proportional to its utilization level when doing useful work, e.g., when the computer system is operating at 30% of its peak performance, it should consume 30% of its peak power. In practice though, power consumption is higher than what the ideal scenario would suggest. Informed by the seminal paper on energy-proportionality by Barroso and Hölzle [46], we define the Energy-Proportionality (EP) of a server as one minus the integral of the relative delta in power consumption with the ideal energy-proportional server, across a range of utilization levels. Referring to Figure 4.1, EP is computed as the area between the server's power consumption and the ideal curve as a function of utilization (area 'A' in Figure 4.1), divided by the area under the ideal curve (area 'B' in Figure 4.1):

$$EP = 1 - \frac{Area\ between\ server\ and\ ideal\ curve}{Area\ under\ ideal\ curve}.$$

An EP of one means that the server consumes power proportional to its load, see top graph in Figure 4.1. An EP of zero means that the server consumes constant amount of power irrespective of its load, see bottom graph in Figure 4.1. The middle graph in Figure 4.1 represents a server with an EP of 0.5; this server consumes 50% of its peak power at zero load. Intuitively, one could think of the EP metric as a quantitative metric for how closely a server's energy-proportionality approaches perfect scaling under a range of server utilization levels.

Figure 4.2 quantifies energy-proportionality over time using the data available on SPEC's power website[1], between the fourth quarter of 2007

[1] http://www.spec.org/power/

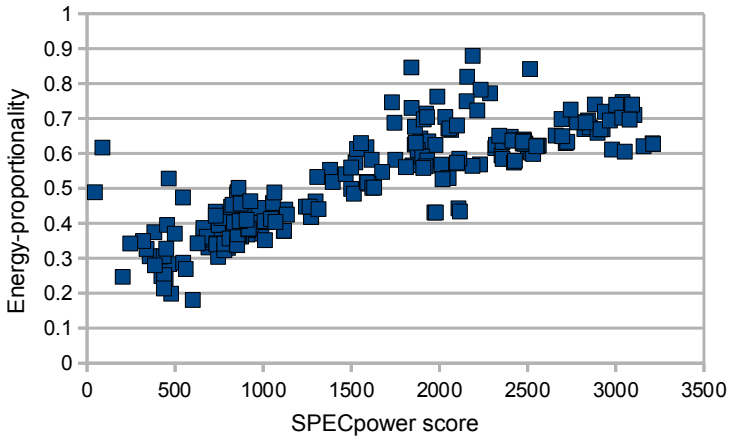

Figure 4.3: Energy-Proportionality (EP) versus SPECpower.

and the first quarter of 2011, as of Jan 10, 2011; this includes 213 systems-under-test in total from 20 vendors. It is encouraging to observe that energy-proportionality has gradually increased over time. Whereas older computer systems (around 2007) had an EP in the 30 to 40 percent range, current systems have an EP in the 50 to 80 percent range; some systems even have an EP close to 90 percent. Clearly, server manufacturers are designing increasingly energy-proportional systems.

4.3 SPECpower and energy-proportionality

As mentioned above, SPEC released the SPECpower benchmark and reports a server's power efficiency using the SPECpower metric, which is defined as the sum of the performance measured at each utilization level divided by the sum of the average power at each utilization level (in intervals of 10 percent), including active idle:

$$SPECpower\_ssj2008 = \frac{\sum Performance}{\sum Power}.$$

Performance is measured as the number of transactions completed per second over a fixed period of time. The benchmark starts its execution with a calibration phase to determine the system's maximum throughput. It then measures performance (throughput) and power at each utilization level starting at maximum load and decreasing in 10 percent increments.

Because SPECpower includes power and performance numbers at different utilization levels, one could argue that SPECpower already includes some notion of energy-proportionality. So, why do we need yet another

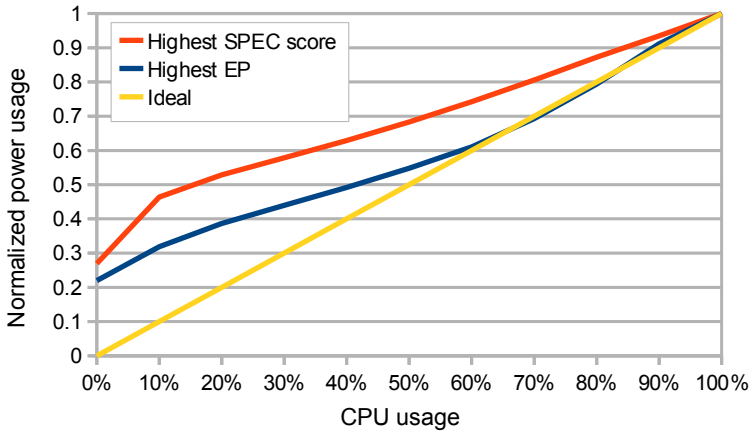

Figure 4.4: Power consumption as a function of CPU load for the systems-under-test with the highest EP and SPECpower scores, respectively.

metric to quantify energy-proportionality? Figure 4.3 plots the 213 systems-under-test in terms of energy-proportionality (EP) versus the SPECpower metric. We observe that SPECpower correlates well with EP, however, the correlation is not perfect. A system with a high EP does not necessarily imply a high SPECpower score, and vice versa. Hence, both metrics have their place. The key difference is that the EP metric focuses on energy-proportionality only and quantifies how a server compares to the ideal energy-proportional system, whereas SPECpower quantifies average power and performance across different server load levels.

4.4 Room for improvement

The EP scores shown earlier demonstrate that energy-proportionality has improved dramatically over time, however, there is yet a gap between contemporary servers and the ideal energy-proportional system. This triggers the question of how much more energy (and cost) can be saved by making servers even more energy-proportional. We consider two servers to address this question: the server with the highest SPECpower score versus the server with the highest EP score. Figure 4.4 shows power consumption at different utilization levels for these two servers. Assuming that servers operate in the 10 to 50 percent range most of the time [46] — in fact, we make the assumption that server operation is uniformly distributed between 10 and 50 percent — we derive how much total energy can be saved by making the server more energy-proportional. For the server with the highest EP score, increasing the machine's energy-proportionality to its ideal can potentially reduce total energy consumption by 34 percent. For

the server with the highest SPECpower score, total energy consumption can be reduced by around 50 percent. We conclude that improving energy-proportionality further may lead to important energy and cost savings.

Although it is hard to expect that we will ever be able to achieve 100 percent energy-proportionality — because there will always be some overhead for keeping a server running — the question of how to improve a system's energy proportionality further remains a relevant and important one. Barroso and Hölzle [46] report that the CPU accounts for 50 percent of total system power at peak performance for a recent Google server. However, at lower utilization levels, the CPU accounts for less than 30 percent. The remaining 70 percent of the total system power is consumed by DRAM, hard drives, power supplies, etc. Looking at a large-scale datacenter, considerable power is consumed in the building's mechanical and electrical infrastructure, such as chillers, computer room air conditioning, UPS systems, power distribution units, humidifiers, etc. This suggests that the CPU is more energy-proportional than the other components in the system. In other words, achieving energy-proportionality at the system level will require improvements across the entire system. Recent research is pursuing this avenue of increasing energy proportionality in the CPU [47], the network [48], and the disk [49]; others advocate for rapidly transitioning an entire server between a high-performance active state and a near-zero-power idle state in response to instantaneous changes in load [50].

Malladi et al. [51] show that the energy proportionality of the memory subsystem can be improved by using mobile DRAM devices for datacenter workloads. Their observation is that these workloads do not use high memory bandwidth but depend on memory capacity and latency. While the mobile DRAM devices have lower peak bandwidth than DDR3, they perform similar to DDR3 with regards to capacity and latency, making them a good match for these workloads.

Wong et al. [52] extended our work on the energy-proportionality metric with two new metrics: linear deviation and proportionality gap. They show that energy proportionality improvements are not uniform across various server utilization levels: the energy proportionality of even a highly proportional server suffers significantly at non-zero but low utilization levels. Using server-level heterogeneity, with an active low-power node and a tightly-coupled high-performance compute node, they enable two energy-efficient operating regions.

4.5 Conclusion

Energy proportionality is a key design target in contemporary servers and datacenters. In this chapter, we quantified the energy proportionality of

contemporary servers. We concluded that energy proportionality has improved significantly over the past few years, from 30 to 40 percent in 2007 to 50 to 80 percent in 2011. Yet, substantial energy savings might be possible to achieve by further improving a server's energy proportionality. Closing the gap between today's most energy-proportional system and the ideal energy-proportional system could potentially lead to an energy (and proportional cost) saving of 34 percent; and energy consumption can be reduced by 50 percent for the machine with the highest SPEC score. Although it is unlikely to expect that we will ever be able to build perfect energy-proportional servers, this analysis suggests that there is ample room for reducing server energy consumption by further improving energy-proportionality.

Chapter 5

Evaluating Computer System Energy Efficiency

A significant limitation with SPECpower, as used in the previous chapter, is that it quantifies energy efficiency for a single, specific workload. As we will see in this chapter, a server's energy efficiency is very much tied to its workload. This chapter therefore proposes SWEEP, Synthetic Workloads for Energy Efficiency and Performance evaluation, a framework for generating synthetic workloads with specific behavioral characteristics. We employ SWEEP to generate a wide range of synthetic workloads while varying the instruction mix, ILP, memory access patterns, and I/O-intensiveness; and we use SWEEP to evaluate the energy efficiency of commercial computer systems across the workload space and learn about how the energy efficiency of a computer system is tied to its workload's characteristics.

5.1 Introduction

Energy efficiency has emerged as a primary design concern across the entire compute range, from low-end embedded systems to high-end servers and datacenters. Embedded systems are typically battery-operated and higher energy efficiency translates into greater user satisfaction through extended battery autonomy. Improving energy efficiency in servers and datacenters reduces the operational cost by reducing the electricity bill for powering the servers as well as for cooling them down. Moreover, there is an environmental concern as well. Improving the energy efficiency of computer systems is key to reduce carbon dioxide emissions by the IT industry.

Architects are well aware of the need for energy-efficient computer systems, and therefore, people have proposed benchmarks and benchmarking methodologies for evaluating energy efficiency, which should ultimately lead to more energy-efficient designs. The Embedded Microprocessor

Benchmarking Consortium (EEMBC) released EnergyBench which provides data on the amount of energy a processor consumes while running a performance benchmark [9]. Recently, SPEC, the Standard Performance Evaluation Corporation, launched SPECpower, a benchmark for evaluating the power and performance characteristics of computer servers [10]. Rivoire et al. [11] propose JouleSort, a sort benchmark aimed at evaluating the energy efficiency of a wide range of computer systems from servers to embedded systems.

Although these approaches offer valuable insight in the energy efficiency of a computer system, they have limited flexibility. The benchmarks are rigid and cannot be altered to reflect different workload behaviors. In particular, EEMBC's EnergyBench is tied to the EEMBC performance benchmarks; the SPEC power benchmark is a Java server workload that generates and completes a mix of transactions; JouleSort implements a sort algorithm. These benchmarks are unable to explore the energy efficiency of computer systems across the workload space. In other words, the numbers produced by these approaches may be limited in scope — they are tied to these specific workloads — and it is hard to generalize towards other types of workloads, i.e., a computer system that is energy-efficient for the power benchmark does not necessarily imply that it is energy-efficient for other workloads.

This chapter proposes SWEEP (Synthetic Workloads for Energy Efficiency and Performance evaluation), a framework for generating synthetic workloads with specific workload characteristics. SWEEP can generate compute-intensive workloads, memory-intensive workloads, I/O-intensive workloads, and any mix thereof. In particular, SWEEP enables its users to configure the workload's characteristics by setting the ratio of integer versus floating-point instructions, the inter-instruction dependencies, memory access patterns, disk I/O access patterns, etc. SWEEP provides a unique opportunity to its users: it allows for exploring the energy efficiency and performance of computer systems by 'sweeping' across the workload space. Using SWEEP we generate a range of synthetic workloads with very different characteristics and run these workloads on two real hardware systems, a low-end system (Intel Atom) as well as a high-end system (AMD Quad-Core Opteron), and evaluate their energy efficiency across different workload behaviors. A preliminary validation using the PARSEC benchmarks and a number of I/O-intensive applications reveals that SWEEP can generate workloads that exhibit a similar performance-energy trade-off as real applications.

We make the following contributions in this work:

- We propose using synthetic workloads for evaluating the energy efficiency of computer systems. In contrast to current power bench-

marking practice which uses specific benchmarks, this chapter proposes a framework, called SWEEP, for generating synthetic workloads with workload characteristics of interest. SWEEP can generate synthetic benchmarks that are compute-intensive, memory-intensive, I/O-intensive, or any mix thereof. By varying the workload characteristics, the SWEEP end user can sweep across the workload space and gain insight in how energy efficiency and performance of a computer system relates to workload characteristics.

- We propose the Energy-Delay Diagram (EDD), a novel visualization method to summarize a computer system's energy consumption and performance relative to a reference machine. EDD clearly illustrates the trade-off in performance versus energy, and provides more insight than the traditional energy-delay-product (EDP) and energy-delay-square-product (ED^2P) metrics.

- Using a wide range of synthetic workloads with very different characteristics, we evaluate the energy efficiency of two real hardware systems, a low-end Intel Atom based machine and a high-end AMD Quad-Core Opteron system. We conclude that for I/O-intensive workloads the low-end machine tends to be more energy-efficient, i.e., it consumes much less energy while achieving similar performance; however, the opposite is true for compute-intensive workloads for which the high-end machine tends to be more energy-efficient: performance is much better and it consumes less or similar total energy. For memory-intensive workloads, there is a trade-off between both.

This chapter is organized as follows. We first revisit prior work in Section 5.2. Section 5.3 presents the SWEEP framework and discusses how we generate a synthetic workload from an abstract workload model. Section 5.4 proposes the Energy-Delay Diagram for evaluating a computer system's energy efficiency. After detailing our experimental setup (Section 5.5), we then use the SWEEP framework to evaluate the energy efficiency of two real hardware platforms using EDDs in Section 5.6. We compare the energy efficiency characteristics of real-life applications and benchmarks against the synthetic workloads and we conclude that the synthetic workloads exhibit a performance versus energy trade-off that resembles the real workloads (Section 5.7). Finally, we conclude in Section 5.8.

5.2 Prior work

5.2.1 Power benchmarks

Given the growing importance of energy efficiency, interest has grown in power benchmarking methods. In the embedded domain for example, EEMBC has released EnergyBench [9], a method for reporting processor energy consumption when running embedded performance benchmarks.

For the server enterprise domain, SPEC recently released SPECpower [10] which is a system-level, server-side Java workload that quantifies energy efficiency under varying loads. SPECpower generates and completes a mix of transactions and the reported throughput is the number of transactions completed per second over a fixed period of time; the workload considers 11 levels of load. Energy efficiency is quantified as the average number of transactions completed per unit of time per Watt.

Rivoire et al. [11] present JouleSort, a sort benchmark that reads its input from a file and writes its output to a file on a non-volatile device. There are three scale categories with 10GB, 100GB and 1TB records, and the benchmark aims at covering multiple domains, from embedded, to mobile, as well as to the server domain. The energy efficiency metric is the total energy consumed by the sort benchmark.

SWEEP is very different in its approach. SWEEP generates synthetic workloads with tunable workload characteristics, which allows for understanding the relationship between energy efficiency of a computer system with respect to workload behavior. The prior power benchmarking proposals are tied to specific benchmarks; SWEEP on the other hand, can generate a range of workload behaviors. Our results, which will be presented later in this chapter, in fact indicate that whether one machine is more energy-efficient compared to another machine is closely tied to its workload: for one workload, system A may be more energy-efficient, whereas for another workload, system B may be more energy-efficient. SWEEP can also be used across multiple domains, from embedded to enterprise.

5.2.2 Synthetic benchmarks

Synthetic benchmarks such as Whetstone [53] and Dhrystone [54] are manually crafted benchmarks that aimed at representing real workloads. Manually building benchmarks though is both tedious and time-consuming. Whetstone and Dhrystone have become less relevant as they no longer represent current workloads.

Statistical simulation [55] collects program characteristics from a program execution and subsequently generates a synthetic trace from it which is then simulated on a simple, statistical trace-driven processor simulator.

The important advantage of statistical simulation is that the dynamic instruction count of a synthetic trace is several orders of magnitude smaller than for today's industry-standard benchmarks, making it a useful simulation speedup technique for quickly identifying a region of interest in a large design space during the processor design cycle.

Recent work proposed automated synthetic benchmark generation [56–58] which builds on the statistical simulation approach but generates a synthetic benchmark rather than a synthetic trace, which allows for running the synthetic workload on an execution-driven simulator as well as on real hardware. Joshi et al. [59] take the idea of synthetic benchmark generation one step further and leverage the synthetic workload generation approach to generate stressmarks or power viruses. They use a genetic algorithm to search the workload space to identify those workload characteristics that maximize average power consumption, peak power consumption, temperature, dI/dt, etc.

This work in statistical simulation and synthetic workload generation has traditionally focused on CPU-intensive workloads, and does not include memory-intensive and/or I/O-intensive behavior. SWEEP on the other hand allows for generating synthetic I/O-intensive and memory-intensive workloads.

Synthetic benchmarks have been developed to evaluate specific aspects of a computer system. For example, the STREAM benchmark seeks at quantifying a computer system's sustainable memory bandwidth using simple vector kernels [60]. IOzone [38] is a filesystem benchmark and generates a variety of file operations. Vasudevan et al. [61] use a set of micro-benchmarks to evaluate the energy efficiency of FAWN (Fast Array of Wimpy Nodes) computing clusters. Gamut, formerly called sstress [62], interleaves the execution of a compute-intensive loop with periods of idleness to match a target CPU utilization, and it also offers possibilities for generating memory-intensive and disk-intensive workloads. SWEEP can generate more diverse workload behaviors in a more flexible way than these specific synthetic (micro)benchmarks.

5.2.3 Energy efficiency metrics

Metrics are at the foundation of experimental research and development. Adequate metrics are absolutely crucial to steer research and development in the right direction. There exist a number of metrics for quantifying a computer system's energy efficiency. Two commonly used energy efficiency metrics are energy-delay product (EDP) and energy-delay-square product (ED^2P) [63, 64]. A major limitation of these metrics is that they combine energy consumption and performance in a single metric, which complicates understanding because in many cases there is a trade-off in

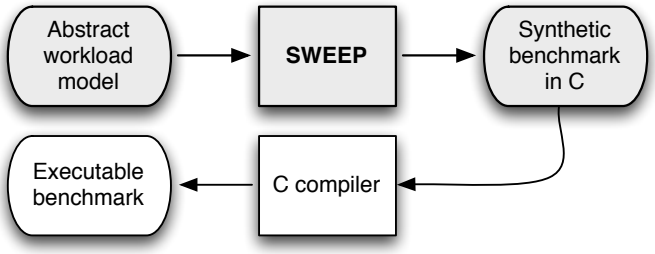

Figure 5.1: High-level view on the SWEEP framework.

performance versus energy, and these metrics may not always capture this trade-off in a comprehensive way, as we will discuss later in more detail. We instead propose EDD, the energy-delay diagram, which visualizes energy consumption versus performance in an insightful way.

Rivoire et al. [11] use total energy consumption as their energy efficiency metric. The winner is the system with the minimum total energy use. While this may be adequate for some workloads, e.g., batch-style background and throughput processes, it is not for performance-critical and latency-sensitive applications such as interactive applications, real-time applications, commercial applications (e.g., Web servers, OLTP), etc. Energy usage by itself may be misleading as an energy-efficiency metric because it does not account for the energy versus performance trade-off. For example, a system that consumes marginally less energy than another system while yielding substantially worse performance is still considered the winner. The EDD instead captures the energy versus performance trade-off.

5.3 SWEEP

5.3.1 High-level overview

Figure 5.1 presents a high-level overview of the SWEEP framework. The end user specifies a set of desired workload characteristics in the abstract workload model from which the SWEEP framework then generates a synthetic workload. The abstract workload is specified in XML format which allows for easily configuring the synthetic workload. SWEEP's output is the synthetic workload, a C program, which is subsequently compiled and run on a simulator or on real hardware.

The concept of the SWEEP framework is such that the workload generator considers a number of building blocks, with each building block representing a different type of behavior. In particular, there is a building block to represent a linear sequence of code (a basic block), a loop, a thread, an access sequence to a data structure in memory, an access sequence to a data

structure stored on disk. These building blocks can be configured at will in terms of their length, their characteristics (e.g., instruction mix, amount of ILP), memory reference locality, etc. For example, the basic block building block specifies the number of instructions, their types and inter-instruction dependencies; the loop building block specifies how many times the loop needs to be iterated; an access sequence to memory specifies the data structure that is to be traversed (array, linked list, tree) and how it is to be traversed.

The SWEEP framework is modular in the sense that it allows for combining these building blocks at will. This allows for building a synthetic workload of interest. For example, one could build a multi-threaded synthetic workload with extensive locking in order to evaluate a particular synchronization primitive. Or, one could build a workload with extensive I/O operations in order to evaluate a system's I/O performance. In this work, we will use the framework to synthesize workloads that are compute-intensive, memory-intensive or I/O-intensive for evaluating a computer system's energy efficiency across these three major classes of workload behaviors.

5.3.2 The SWEEP building blocks

There are five building block types in total, which we briefly discuss now.

Basic block

The 'basic block' building block represents a linear sequence of instructions and is an atomic unit of work. The basic block can be configured through a number of parameters, such as the number of instructions in the basic block, their types (integer or floating-point) and their inter-instruction dependencies. The latter determines the amount of instruction-level parallelism (ILP) in the program. The inter-instruction dependency distance is defined as the number of dynamically executed instructions between writing a data value and reading it. Hence, a large inter-instruction dependency distance implies high ILP, and a small dependency distance implies low ILP.

An additional parameter specifies the probability for the basic block to be executed. This is useful for generating conditional control flow in the synthetic workload (e.g., if-then-else statement).

Loop

The 'loop' building block specifies that the enclosed building blocks need to be iterated a number of times. The number of iterations is to be set by

the SWEEP end user. The loop building block can include other loop building blocks which allows for building nested loops of any depth. Also, it can include basic blocks that the loop will iterate on, and if the basic blocks have conditional execution probabilities associated with it, then the generator will generate conditional control flow within the loop (e.g., if-then-else statements with a hard-to-predict branch within the loop).

Memory

The 'memory' building block specifies a memory-intensive program sequence. The main attribute specifies the data structure and its size that is to be accessed; there are three options: an array, a linked list and a binary tree. Also, there are a number of possible access patterns. For the array, one can have a sequential, strided or random access pattern; for the linked list, the only access pattern is to sequentially traverse the linked list; for the tree data structure, the end user has the ability to select a breadth-first or depth-first access pattern. These access patterns can be either reads or writes, and are initiated within a loop.

Multi-threading

There are three building blocks related to multi-threaded execution. (1) The 'thread' building block initiates a thread in the synthetic workload. An attribute of the thread building block is whether the data structures accessed within the enclosed memory building blocks are private (access by the given thread only) or global (accessed by all threads). (2) The 'thread group' building block can be used inside the loop building block and allows for initiating parallel work done by threads that join (barrier synchronization) before proceeding to the next iteration. (3) The 'mutex' building block specifies that the enclosed building blocks are part of a critical section and thus need synchronization using locks.

Input/Output

Finally, the 'I/O' building block initiates reads and writes to a file stored on disk. There are three attributes: (1) the size of the file, (2) the access pattern (sequential, strided or random), and (3) whether the file is to be read or written. In order to fully stress the disk, there is an option to eliminate the buffering by the operating system and disk.

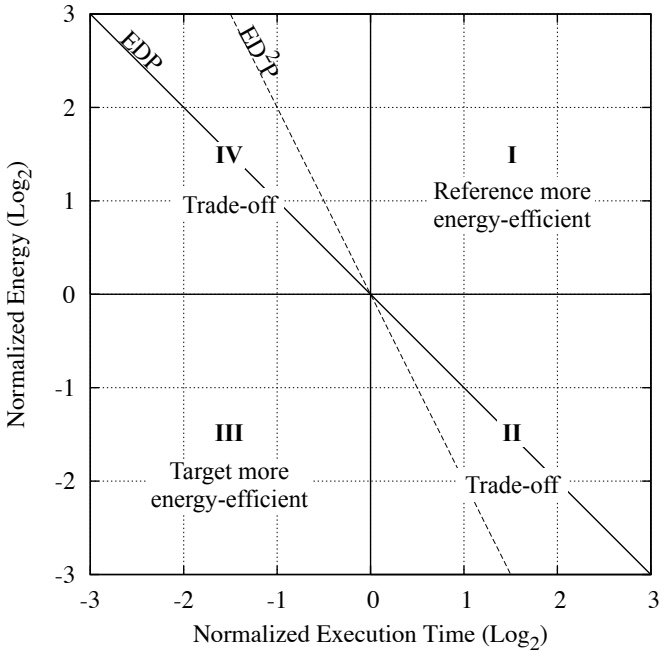

Figure 5.2: Energy-Delay Diagram.

5.4 Energy-Delay Diagram

As mentioned in the introduction, we use the SWEEP framework to generate different flavors of workload behaviors in order to evaluate a computer system's energy efficiency. Now, quantifying energy efficiency is by itself a non-trivial issue. Traditionally, two metrics are being used for evaluating a computer system's energy efficiency, namely energy-delay product (EDP) and energy-delay-square product (ED^2P). EDP is defined as the total energy consumed to execute a unit of work multiplied by the execution time; ED^2P is defined as energy multiplied by the square of the execution time — hence, ED^2P puts more emphasis on performance than EDP. EDP and ED^2P are appealing because they quantify energy efficiency by a single number. However, evaluating a computer system's energy efficiency by a single metric may be misleading or at least it may complicate understanding the energy versus performance trade-off.

The Energy-Delay Diagram (EDD) visualizes the energy versus performance trade-off in an intuitive way, see Figure 5.2. The vertical axis shows the logarithm of the ratio of the energy consumption on the target machine relative to the reference machine:

$$y = \log_2 \left(\frac{Energy_{target}}{Energy_{reference}} \right). \tag{5.1}$$

The horizontal axis shows the logarithm of the ratio of the execution time on the target machine relative to the reference machine:

$$x = \log_2 \left(\frac{Time_{target}}{Time_{reference}} \right). \tag{5.2}$$

The origin of the EDD represents the reference machine. The first quadrant (I) represents cases in which the reference machine is more energy-efficient than the target machine, i.e., the reference machine consumes less energy and execution time is shorter. The third quadrant (III) represents the opposite situation: the target machine is more energy-efficient than the reference machine, i.e., the target machine consumes less energy and yields better performance. The second (II) and fourth (IV) quadrants represent trade-offs. For example, in quadrant II, the reference machine yields better performance at the cost of consuming more energy; in quadrant IV, we have the dual situation: the target machine yields better performance at the cost of consuming more energy. An important feature of the EDD is that, because it uses the logarithm of the energy and performance ratios, the EDP and ED^2P metrics can be visualized as straight lines in the EDD. The EDP line, which denotes points where the target and the reference machines are equally energy-efficient according to the EDP metric, is shown as the anti-bisector in Figure 5.2; the ED^2P line is shown as well.

The EDD visualizes the energy efficiency trade-off in an intuitive way. For example, a target system that is equally energy-efficient as the reference machine according to the EDP metric will appear on the anti-bisector. If the target system appears in quadrant II (on the EDP line), this means that the target system consumes less energy at the cost of a proportional loss in performance; if it appears in quadrant IV, this means that the target system consumes more energy at the benefit of a proportional performance gain. As another example, a target system appearing above the EDP line in quadrant II, implies that the reference system is more energy-efficient than the target system according to the EDP metric; however, the EDD shows that there is a trade-off: the reference system consumes less energy, but this comes at a performance hit (however, the performance hit is relatively small compared to the reduction in energy). In other words, the EDD clearly illustrates the trade-off in energy consumption versus performance.

Use case #1:
Comparing machines for a fixed workload. One possible use case for EDDs is to visualize the energy and performance trade-off of computer systems. For example, plotting different machines in the EDD enables a quick and intuitive competitor analysis in terms of the energy efficiency of computer systems for a given benchmark or a set of benchmarks. Figure 5.3 shows an illustrative EDD with four machines, A (the reference machine),

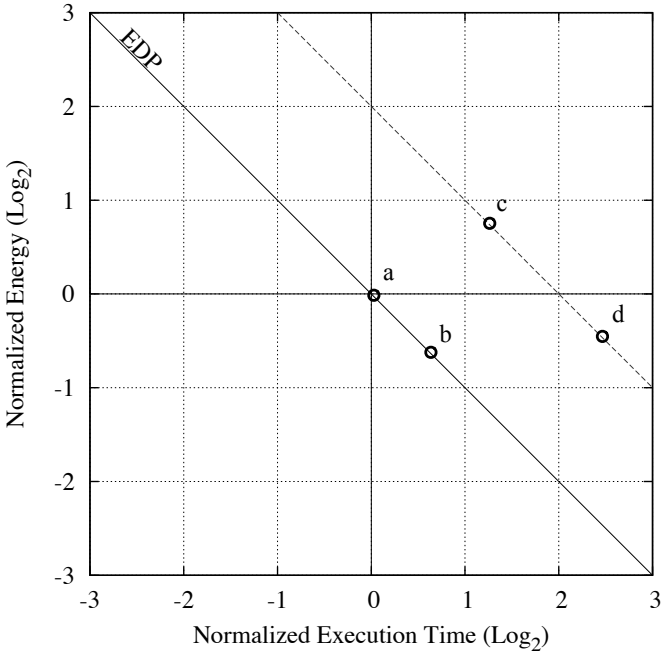

Figure 5.3: Comparing machines' energy efficiency using the EDD.

B, C and D. Machine B achieves the same EDP as machine A as they lie on the anti-bisector EDP line. Also, C and D achieve the same EDP as they lie on a straight line parallel with the EDP line. This is a key feature of the EDD: design points that achieve the same EDP lie on a straight line parallel with the EDP line — this is a result of representing the logarithm of energy and execution time on the vertical and horizontal axes.

In this same example, machine C is less energy-efficient than both A and B: energy consumption is higher and performance is lower. Machine D on the other hand represents a trade-off relative to A: D consumes less energy than A at the cost of delivering worse performance.

Use case #2:
Comparing machines across workloads. Another use case, which we will explore further in this chapter, is to consider two computer systems (a reference and a target machine) and a range of workloads, and then provide data points for each of the workloads in the EDD. This enables exploring whether the energy efficiency of one system compared to another is subject to the workload. And given the SWEEP framework, this will enable us to explore how energy efficiency of a computer system relates to workload characteristics.

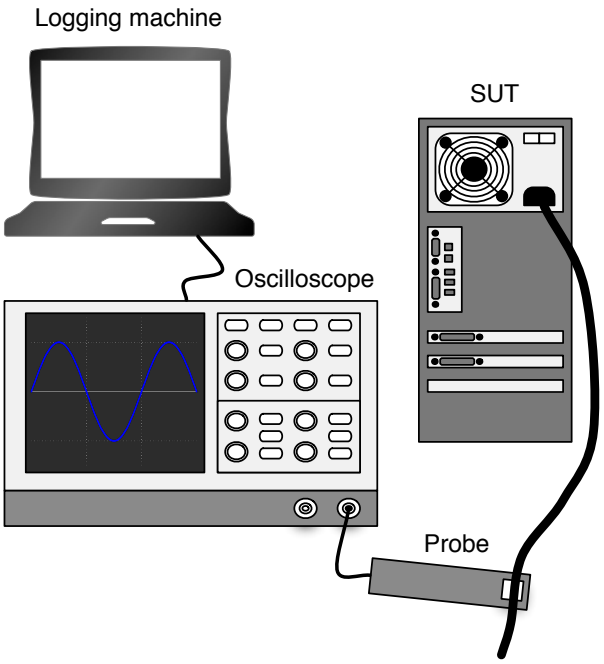

Figure 5.4: Runtime power monitoring setup.

5.5 Experimental setup

Figure 5.4 illustrates our runtime power monitoring setup. The probe (Tektronix TCP 202) of an oscilloscope (Tektronix TDS 7104) is connected to the power cord of the System Under Test (SUT). The probe measures the current flowing through the power cord which enables measuring the total power consumed by the SUT. The oscilloscope is connected to a logging machine, which allows for post-processing the experiment data. This setup is similar to the one used by others [65].

We consider two SUTs in our experiments, a low-end Intel Atom machine and a high-end AMD Quad-Core Opteron server, see Table 5.1. The Intel Atom processor is a dual-core processor. Each core is an in-order SMT core with two thread contexts. The cache hierarchy is private to each core. The AMD Opteron is a quad-core processor. Each core is a superscalar out-of-order core (without SMT). The L1 and L2 caches are private and the L3 cache is shared among the cores. Both machines have a comparable 7200 rpm hard disk. The Thermal Design Power (TDP) is very different: the TDP for the Intel Atom is rated to be 8 Watt whereas the TDP for the AMD Opteron is rated to be 95 Watt.

Low-end Intel Atom machine	
CPU	1.6 GHz Intel Atom 330 2 cores, 2 SMT threads per core 56 KB private L1, 512 KB private L2 TDP: 8 Watt
Memory	DDR2-800, 2 GB
Disk	WD Scorpio Blue 7200 rpm
Power supply	Antec Trio 550 (85% efficiency)
High-end AMD Quad-Core Opteron	
CPU	2 GHz AMD Opteron 2350 Barcelona 4 cores 128 KB private L1, 512 KB private L2, 2 MB shared L3 TDP: 95 Watt
Memory	DDR2-667, 4 GB
Disk	Samsung SATA 7200 rpm
Power supply	Antec EA 380D Green (80% efficiency)

Table 5.1: The Systems Under Test: a low-end and a high-end machine.

5.6 Real system evaluation

We now exploit the unique property offered by SWEEP to 'sweep' the workload space and gain insight in how the energy efficiency of a computer system is affected by the characteristics of its workload. We systematically vary workload characteristics in the abstract workload, generate synthetic workloads, and run these synthetics on both of our SUTs. In the EDDs to follow, we consider the high-end AMD Opteron server as the reference machine. We organize the discussion along three major flavors of workload types: CPU-intensive, memory-intensive and I/O-intensive workloads.

5.6.1 CPU-intensive workloads

The first synthetic workload that we generate is a compute-intensive workload. It involves a limited number of memory accesses (and all memory accesses are cache hits), and performs no disk I/O. The workload consists of floating-point operations and the workload characteristic that we vary here is the inter-instruction dependency distance. An inter-instruction dependency distance of one means that an instruction is dependent on the instruction before it in the dynamic instruction stream. In other words, the synthetic workload involves a long chain of dependent instructions, and hence, there is no ILP. Increasing the inter-instruction dependency distance increases the opportunities for exploiting ILP and hence, performance improves.

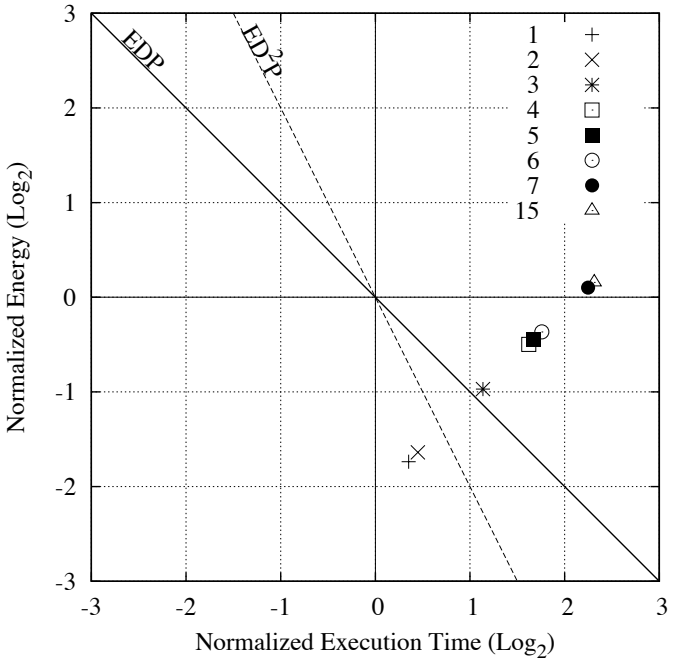

Figure 5.5: EDD for a CPU-intensive workload with varying inter-instruction dependency distance (see legend).

Figure 5.5 shows the EDD for the CPU-intensive workloads for a varying inter-instruction dependency distance (see the legend). A workload with no or limited ILP (i.e., a short inter-instruction dependency distance of 1 or 2) is more energy-efficiently run on the low-end machine than on the high-end machine, according to both the EDP and ED^2P metrics: the points corresponding to an inter-instruction dependency distance of 1 and 2 lie under the EDP and ED^2P lines. At higher degrees of ILP, the high-end server is more energy-efficient: the points lie above the EDP and ED^2P lines. And for high degrees of ILP (inter-instruction dependency distance of 7 and higher), the high-end machine clearly is the most energy-efficient machine: it consumes less total energy and execution time is shorter. This result can be explained by the fact that the high-end machine is a superscalar out-of-order processor which can better exploit the available ILP than the low-end in-order processor can. Clearly, for the high-end processor and workloads with high levels of ILP, the shorter execution time outweighs the higher power consumption of the processor, which ultimately leads to an overall reduction in the total amount of energy consumed. The interesting observation is that compute-intensive, high-ILP workloads are more energy-efficiently run on high-end processors, i.e., high-end processors yield better performance at lower total energy use.

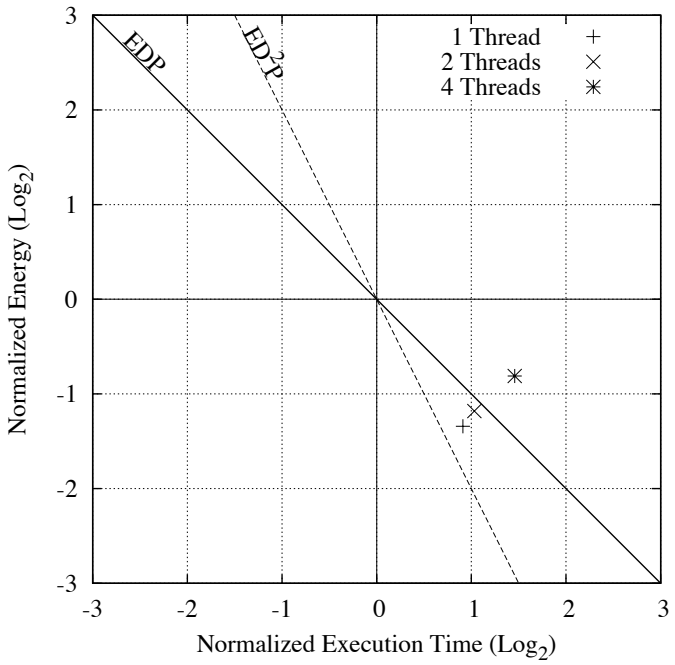

Figure 5.6: EDD for memory-intensive, multi-threaded workloads.

5.6.2 Memory-intensive workloads

Our next experiment considers memory-intensive multi-threaded workloads. These workloads access a 150 MB binary tree, and each thread accesses a private tree. All threads perform a breadth-first tree search and read all the values along the tree (over 85% of the instructions are loads). The IPC (for a single thread) on the high-end AMD Opteron processor is fairly low, namely 0.24. The reason is twofold: relatively high cache miss rates in the L2 and L3 caches, and low branch prediction accuracy. Figure 5.6 shows that both machines are comparable in terms of their energy efficiency according to the EDP metric (for one thread and two threads). For four threads, the high-end quad-core processor is more energy-efficient compared to the low-end dual-core (two-way SMT per core) processor. The reason is the more aggressive memory hierarchy of the high-end processor (more on-chip cache space and more memory bandwidth) along with the fact that each thread on the high-end machine runs on a private core. On the other hand, the low-end machine's memory hierarchy is less aggressive, and two SMT threads per core share many of the resources. The memory system performance advantage of the high-end processor outweighs the additional energy consumed by the additional cores. The important observation here is that there is a trade-off in energy versus performance for memory-intensive workloads: the high-end processor yields better perfor-

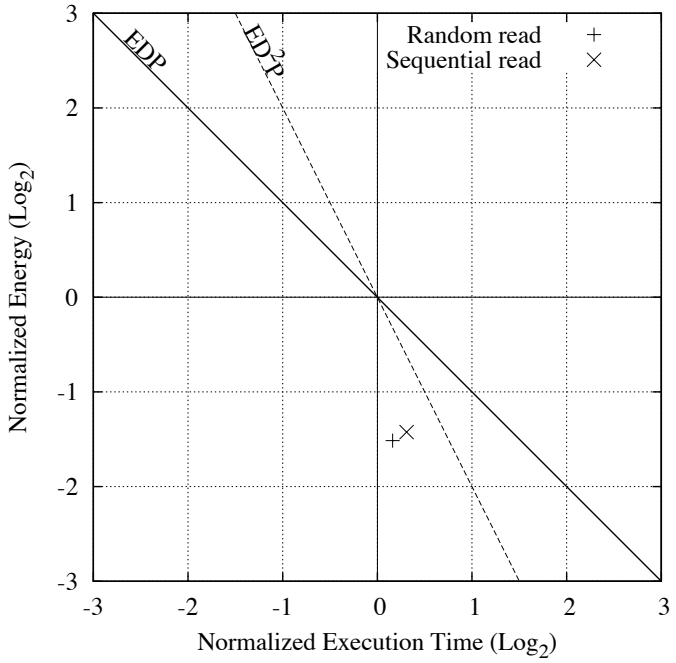

Figure 5.7: EDD for I/O-intensive workloads.

mance at the cost of consuming more energy; the low-end processor on the other hand consumes less total energy but performance is worse.

5.6.3 I/O-intensive workloads

For I/O-intensive workloads that read randomly or sequentially through a 12GB file, the low-end processor tends to be more energy-efficient than the high-end processor according to both the EDP and ED^2P metrics, see Figure 5.7. The low-end processor yields slightly less performance than the high-end processor, however, it consumes much less energy. The reason is that the processor is waiting for the disk to return while it is consuming power, and since the high-end processor is consuming more power than the low-end processor, the end result is that the low-end processor is more energy-efficient for this type of workloads. This also explains why the low-end processor is relatively more energy-efficient for the random read access pattern than for the sequential read pattern, i.e., the random access patterns introduces even more wait time for the processor than the sequential access pattern does.

5.7 Real-life applications

So far, we have considered synthetic workloads only. We now consider real applications (both benchmarks and GNU programs) and we evaluate whether the real applications lie in a region that is comparable to the region covered by the synthetics. In other words, we want to do some preliminary validation to gain confidence with respect to whether the synthetic workloads generate a performance versus energy trade-off that somehow relates to real application behavior. It is not our intent to validate that SWEEP can generate synthetic workloads that can serve as proxies for real-life applications, rather we want to evaluate whether the conclusions we obtained in the previous section using synthetics hold true when considering real applications.

Our first set of applications is taken from the multi-threaded PARSEC benchmark suite [15]. We consider four benchmarks, freqmine, raytrace, swaptions and streamcluster, see Figure 5.8; each benchmark runs four threads. The high-end machine is clearly more energy-efficient than the low-end machine for these benchmarks. Especially for streamcluster, the high-end machine yields better performance and consumes less energy; for swaptions, the high-end machine yields better performance at the same energy as the low-end machine. For the other benchmarks, freqmine and raytrace, there is a trade-off, however, the high-end machine is more energy-efficient according to both the EDP and ED^2P metrics. This result suggests that the PARSEC benchmarks are primarily CPU-intensive, exhibit substantial ILP and have limited memory requirements.

Our second set of applications comprises well-known GNU tools, namely tar and gzip. The tar tool creates an archive and is I/O-intensive: it reads a number of files and writes them in an archive, the tarfile. The second tool combines tar with gzip: it tars a number of files and then compresses it in a gzipped tarfile. We consider two compression levels here: 1 and 5 (5 means higher compression than 1). Figure 5.9 shows the EDD for the tar and gzip applications. Interestingly, the low-end machine is more energy-efficient for the tar workload, whereas the high-end machine is more energy-efficient for the tar+gzip workload. The reason for this difference is that the tar workload involves I/O operations almost exclusively, whereas the tar+gzip workload also involves substantial CPU-intensive operations during compression. This is further explained by the observation that the high-end machine is even more energy-efficient for gzip's CPU-intensive compression level 5 than for compression level 1.

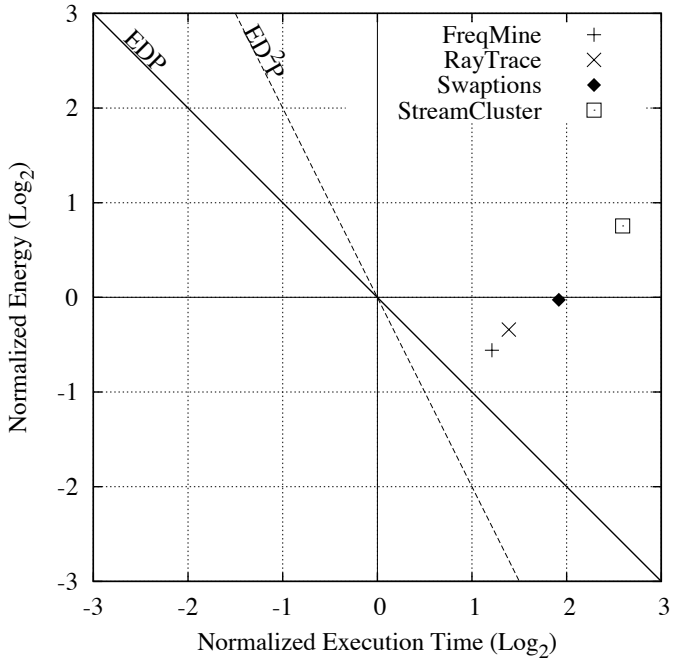

Figure 5.8: The EDD considering some of the PARSEC benchmarks.

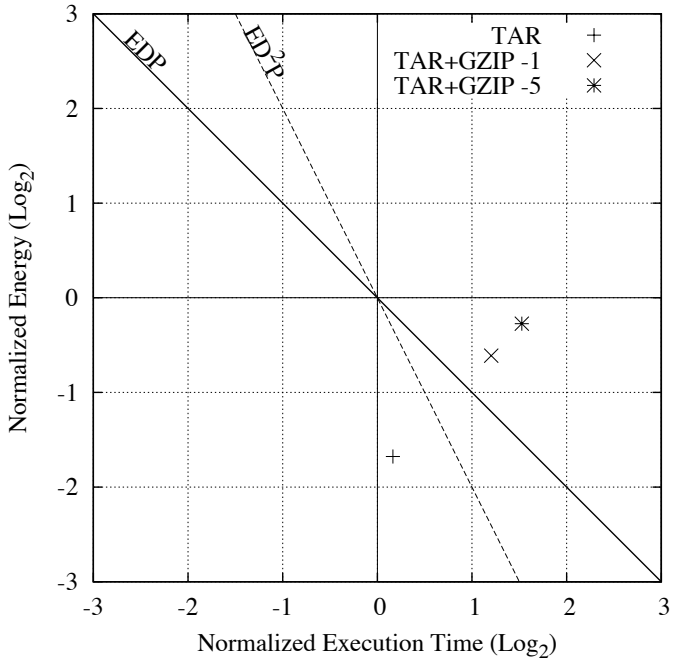

Figure 5.9: The EDD considering the tar and gzip Linux tools.

5.8 Conclusion

This chapter proposed SWEEP, a framework for generating synthetic workloads with specific behavioral characteristics. SWEEP can generate compute-intensive, memory-intensive and I/O-intensive workloads, and any mix thereof. SWEEP enables novel capabilities to study a computer system's energy efficiency. Whereas prior work in power benchmarking is tied to specific benchmarks, such as EnergyBench, SPECpower and JouleSort, SWEEP enables sweeping the workload space and study how energy efficiency is tied to the workload characteristics. We conclude that whether one machine is more energy-efficient than another machine is very much workload dependent. SWEEP is a useful tool to explore these trade-offs.

This chapter also presented the Energy-Delay Diagram (EDD), a novel way of visualizing a machine's energy efficiency relative to a reference machine. The EDD represents the trade-off in performance versus energy in a more intuitive way than the traditionally used EDP and ED^2P metrics do. Throughout this chapter we have used EDD to visualize the performance versus energy trade-off for real hardware platforms. However, these diagrams can also be used to visualize results from simulation. Using our interval simulator in COTSon (Chapter 2) combined with the McPAT [66] power model, we would be able to visualize the performance versus energy trade-off for future hardware platforms.

We believe this work points towards an interesting avenue of future work. The observation that some workloads are more energy-efficiently run on one machine whereas other workloads are more energy-efficiently run on another machine, suggests that heterogeneous datacenters may be an energy-efficient solution. In a heterogeneous datacenter, workloads would be steered dynamically towards the most energy-efficient server. Given the trend towards cloud computing which suggests many different workloads running in consolidated environments, there may be opportunities for exploiting workload diversity in the datacenter for improving overall energy efficiency and decreasing (operational) cost.

Chapter 6

Analyzing Long-Tail Latencies

Having considered simulation and real hardware measurements in a test environment, we now consider performance issues in a real-life production datacenter. This chapter proposes TPA, a linear-temporal logic based language for tracking performance problems that cause long-tail latencies in large online datacenter workloads. Since interactions between operations are often responsible for long-tail latency, we must analyze fine-grained traces to investigate their cause. Given these formulas, our system searches through traces to find matches for these formulas and extracts relevant information from the matches.

6.1 Introduction

Web services, such as search, work on many user operations in parallel. Thus, the time to service a user operation depends not just on the nature of the operation but also on the availability of resources. For example, an operation that normally takes 1 ms may take as long as 1 second if some other operation is holding a resource (e.g., lock) that it needs. Indeed, this contention for resources often causes the dreaded *long-tail latency* that afflicts web services. Long-tail latency directly affects capacity planning and user experience for Web services [1]; thus, it is critical to understand and improve. Unfortunately, to analyze long-tail latency we must reason over fine-grained *traces* (i.e., timestamped sequence of events) which is laborious and requires extensive domain knowledge. This chapter describes how to use formulas in linear-temporal logic [67] extended with variables to analyze traces and thus reduce the manual labor involved in analyzing traces.

Reasoning with traces is difficult because we must reason across chains of events where each event affects subsequent events. For example, let's suppose we wish to find how often a high-priority thread, e.g., H, waits for a lock that a low-priority thread, e.g., L, holds (i.e., a priority inversion). Given a kernel trace, we can readily find when H waits on a lock (it shows

up as system calls to *futex* in the Linux kernel trace[1]). To determine if H was waiting for a lock held by L we must now reason across time to determine if L released the lock just before H acquired it. Using manual effort and search tools we can do this reasoning for a few operations. However, reasoning over a few operations does not tell us if the priority-inversion is common or rare. Thus, we must do this reasoning over many long traces.

To enable such reasoning in our system (TPA), experts express their knowledge in our extension to linear temporal logic (LTL). For example, an expert can express the formula: (i) L acquires a lock at time t_0; (ii) then, H blocks on a call to *futex* at time t_1; (iii) then, L eventually releases the lock at time t_2; (iv) then, L eventually issues a wakeup for other threads waiting on the lock at time t_3; (v) then, H eventually wakes up from waiting on the lock at time t_4. TPA matches this formula against traces and each match produces a binding for the timestamps (i.e., the t_i); we can use these timestamps to determine the lock holding time for thread L and the blocked time for thread H.

The main contributions of this work are to show that (i) we can extract useful and actionable information from performance traces using simple LTL formulas; (ii) LTL is an intuitive notation for expressing patterns in traces (i.e., the LTL formulas follow naturally from an understanding of the sequence of events that we wish to match); and (iii) our implementation scales to large traces.

The remainder of this chapter is organized as follows. Section 6.2 motivates the need for TPA. Section 6.3 describes our notation for expressing formulas in LTL. Section 6.4 describes our algorithm for matching formulas to traces. Section 6.5 describes optimizations to TPA. Section 6.6 evaluates TPA. Section 6.7 reviews related work and Section 6.8 concludes.

6.2 Motivation

To enable performance (or other) debugging, many applications produce traces in the form of logs. For example, an electronic banking service may write out a log line whenever any of the following events happen:

1. RequestReceived[Timestamp=$\cdots$, User=$\cdots$, Id=$\cdots$]
2. GetAccountNumber[Timestamp=$\cdots$, User=$\cdots$, Id=$\cdots$]
3. GetBalance[Timestamp=$\cdots$, User=$\cdots$, Id=$\cdots$]
4. ResponseSent[Timestamp=$\cdots$, User=$\cdots$, Id=$\cdots$]

[1]Even this is non-trivial since the *futex* call is used in many ways, only some of which indicate contention.

Each event has a name (e.g., "RequestReceived") and three properties: a timestamp, the user making the request, and a unique identifier for the request.

A successful request produces the sequence of events (1, 2, 3, 4). An unsuccessful request produces the sequence of events (1, 2, 4); i.e., a failing request does not get the account balance. To quantify the time spent in successful and unsuccessful requests, we must reason over sequence of events using a state machine: different sequences *mean* different things.

Now, let's suppose users may have many requests in flight at the same time. A request that runs in isolation has different performance characteristics from a request that runs in parallel with other requests. Thus, we wish to further refine our timings into: (i) successful in isolation; (ii) successful with others; (iii) unsuccessful in isolation; and (iv) unsuccessful with others. This is difficult to calculate because we must reason over not just linear sequences of events but also interactions between interleaved sequences.

For these examples, we could have obtained the data by modifying the banking system either manually or using an semi-automated system, such as DTrace [68] or aspect-oriented programming [69]. This would be inacceptably slow in our environment: all production software must undergo a thorough reviewing and testing process before it can be deployed and thus we would have to wait for weeks or more to get the data we need.

Unfortunately, getting the data by analyzing these traces is difficult since we must reason across many long and interleaved sequences of events. Before developing TPA, we used off-the-shelf tools, such as regular expression matchers, but found them lacking for two main reasons: (i) they are inconvenient for finding *all* interleaved instances of a pattern (but can often find a single instance); and (ii) require significant coding for expressing complex patterns (e.g., *ResponseSent* must occur within 10 ms of *RequestReceived*). The remainder of this chapter describes TPA and shows that it enables us to compactly express patterns and match them efficiently against traces.

6.3 Expressing formulas using temporal logic

Our goal with this work was *not* to *design* a new language but to *use* an existing well-known notation for solving our concrete problems. Consequently, our language is a minimal extension to LTL.

6.3.1 Definitions: Traces, formulas, and events

A *trace* is a sequence of timestamped events in time order. For example, $(e_i e_{i+1} \cdots e_j)$ is a trace whose events are e_i, e_{i+1}, $\cdots$, e_j and e_i occurs before

e_{i+1} which in turn occurs before e_{i+2} and so on. Each event has a timestamp[2] and optional attributes. Each attribute is a key-value pair; where key is the name of the attribute and value is either a string or an integer value.[3] For example

syscall[Timestamp=10, name="write"]

is an event with name "syscall" at time 10 and has the attribute "name" with value "write" (similarly to EBBA [70]). A *formula* is a pattern that TPA can match against traces.

6.3.2 Event formulas

Event formulas match a single event:

- e: Matches any event with name e.
- $e[a_1 = v_1, a_2 = v_2, \cdots]$: Matches events with (i) name e and (ii) attribute a_1 having value v_1, a_2 having value v_2, and so on. A matching event must have the specified attributes but it may have additional attributes.

In addition to $=$ which matches attributes exactly, we support the following operators: ($\neq$) for attributes of type integer or string, and ($>$), ($\geq$), ($<$), and ($\leq$) for integer attributes. We support a special event name "*" which matches all event names, this enables us to match attributes without matching a specific event name.

We can combine formulas using boolean operators, And ($\wedge$), Or ($\vee$), and Not ($\neg$).

6.3.3 Trace formulas

Trace formulas match a sequence of events (i.e., traces). For example, let's describe the formula for the unsuccessful case in Example 1 (Section 6.2):

The first event for User *u* and Id *i* is *RequestReceived*, and
the *next* event for *u* and *i* is *GetAccountNumber*, and
the *next* event for *u* and *i* is *ResponseSent*

[2]For most of our traces, "timestamp" is in microseconds since epoch. However, TPA does not care what time base we use; for example, we could use an incrementing counter as a timestamp.

[3]These are the only types of attributes that we needed; it is trivial to extend our system for other types of values.

Intuitively, this formula contains three (sub-)formulas for matching the three events of the sequence, connected by *And*, and sequenced by the temporal operator *Next*. A given trace may contain numerous interleaved requests; to find all such instances, we must match the above formula to *all* points of a trace. Consequently, we define the temporal operators in terms of subtraces.

An event formula matches a subtrace if it matches the first event in the subtrace. In addition, we support the following three LTL operators:

1. $\bigcirc f$ (read as *Next* f) matches a subtrace $(e_i e_{i+1} \cdots e_j)$ if f matches $(e_{i+1} \cdots e_j)$. For example "$\bigcirc$syscall" matches a subtrace whose second event has the name "syscall" and the first event can be any event, including "syscall".

2. $\Diamond f$ (read as *Eventually* f) matches a subtrace $(e_i \cdots e_j)$ if f matches the full subtrace or any suffix of the subtrace (i.e., $(e_{i+n} \cdots e_j)$ where n is a non-negative number and $i + n \leq j$). For example "$\Diamond$syscall" matches a subtrace that has an event with name "syscall" somewhere in the subtrace.

3. $f \ \mathcal{U} \ g$ (read as f *Until* g), where f is an event formula (i.e., f needs only a single event to match) and g is any formula, matches a subtrace $(e_i \cdots e_j)$ if we can split the subtrace into two parts (a) $(e_i \cdots e_k)$ and (b) $(e_{k+1} \cdots e_j)$ such that (i) f matches every suffix of (a) but g does not match any suffix of (a); and (ii) g matches (b). (a) may be empty and f may or may not match (b). For example: "E1 $\mathcal{U}$ E2" matches the subtraces (E1, E1, E2) and (E2). We restrict f to an event formula (and not a temporal formula) because it must reduce to true or false for every event in (a).

In addition to the above three temporal operators, we find a constrained variation of $\bigcirc$ to be indispensable: "$\bigcirc[a_1 = v_1, \cdots]f$" matches a subtrace $(e_i e_{i+1} \cdots e_j)$ if f matches the first event in the subtrace $(e_{i+1} \cdots e_j)$ that has attributes $a_1, \cdots$ with corresponding values $v_1, \cdots$. Constrained $\bigcirc$ is not a primitive LTL operator: we can express it using the basic $\bigcirc$ and $\mathcal{U}$ as follows:

$\bigcirc[a_1{=}v_1, \cdots] \ f = \bigcirc(\neg^*[a_1{=}v_1, \cdots] \ \mathcal{U} \ (^*[a_1{=}v_1, \cdots] \wedge f)$

Intuitively, to express the constrained *Next*, we skip any events that do not have the matching attributes ($\neg^*[a_1{=}v_1, \cdots]$) until we find the first event that matches ($^*[a_1{=}v_1, \cdots]$). At that matching event, f should also match.

6.3.4 Variables

Let's try to write a formula that matches "all lock operations with their corresponding unlock operations":

lock[pid=102] $\wedge\bigcirc$[pid=102] unlock

This formula matches the lock/unlock pattern for process-id 102. The $\bigcirc$ finds the first event after the lock for the process-id 102 and makes sure that it is an unlock. To generalize such formulas, we allow the "values" in the constraints to be variables; we bind the variables when we match the formula; each variable must have a single binding which the user specifies using "$\rightarrow$". Using this, we can write our pattern as:

lock[pid $\rightarrow$P] $\wedge\bigcirc$[pid=P] unlock

Using $\rightarrow$ instead of $=$ binds the pid of the lock event to P. We can now use P elsewhere in the formula to mean the bound value. A formula can only use bound variables.

6.3.5 Extended example

We now give an extended example to show how the different operators work together. This example is based on the following real world situation: the C++ mutex implementation at Google uses a queue to hold all threads waiting on the mutex. If a thread needs to block on a mutex, it must put itself on the mutex's "waiting" queue; a spin-lock guards access to this queue. If the thread cannot get the spin lock after a few iterations of spinning, it goes to sleep for k nanoseconds using a call to *nanosleep*; on wakeup it tries to acquire the spin lock again. The worst manifestation of this situation is when the thread holding the mutex cannot access the waiting queue to find which threads to wake up when releasing the mutex; thus, the thread's inability to acquire the spinlock results in a longer holding time for the mutex which further aggravates contention. The mutex implementers did not expect this situation to ever happen and consequently did not include instrumentation to count its occurrences. Section 6.6.5 uses TPA to show that it happens surprisingly frequently. Given a kernel trace for a thread, we can express this situation intuitively as:

1. The thread is executing in user space, and *next*
2. The thread calls nanosleep, and *next*
3. The kernel preempts our thread and gives the CPU to another thread (i.e., context switch), and *next*

4. The kernel gives the CPU back to our thread (i.e., another context switch) and, *next*
5. The thread wakes up from nanosleep, and *next*
6. The thread continues executing in user space, and *next*
7. The thread calls *futex* to release the lock, and *next*
8. The thread calls *sched_wakeup* to wake up the waiting threads.

Our actual formula closely mirrors the above intuition:

RunInUserSpace[thread_id →t]
∧ ○[thread_id=t] (syscall[name="nanosleep"]
∧ ○[thread_id=t] (preempt
∧ ○[thread_id=t] (resume
∧ ○[thread_id=t] (syscall[name="nanosleep"]
∧ ○[thread_id=t] (RunInUserSpace
∧ ○[thread_id=t] (syscall[name="futex"]
∧ ○[thread_id=t] sched_wakeup))))))

This formula uses the constrained next to make sure that all events in the sequence are from the same thread; between these events the trace may contain concurrent events for other threads. Specifically, it uses a variable (t) to hold the thread id for a thread and the constrained next skips over events for threads other than t.

6.4 Matching formulas

Prior work has mostly used LTL for verifying systems [71,72]: i.e., the goal is to determine if a given LTL formula is satisfiable. In contrast, we wish to (i) determine whether or not a particular formula matches a *particular* trace; and (ii) extract information from the matches.

A formula reduces to true when it *matches* a trace. A formula that has not yet reduced to *true* or *false* is *pending*; i.e., it needs to see additional events before it reduces to *true* or *false*. We match one or more formulas in a single traversal of the trace.

A given formula may match at many points in a trace; thus we create formula *instances* for each possible match. Each instance records everything that it needs, including its environment, to determine a successful or unsuccessful match. While traversing a trace, we create a new instance for each formula at each event in the trace. We match all pending instances (including the ones just created) to each following event in the trace. Matching a formula instance to an event may reduce the instance to true, false, or a pending state. Once an instance reduces to the true or false state, it stays in

that state no matter what further events we match it to. We now give the reduction rules for each construct.

6.4.1 Variable bindings

Many formulas contain sub-formulas; e.g., the formula $f \wedge g$ has two sub-formulas (f and g) which in turn may have sub-formulas of their own. A sub-formula instance can use the variable bindings in its parent instance to do its matching. When a sub-formula reduces to true, its bindings become part of its parent's binding; after this point other sub-formulas of the parent can also use these bindings.

6.4.2 Event formula

An event formula instance, such as "event1[attr=3]" reduces to true if it matches the first event of the subtrace and false otherwise. Furthermore, on reducing to true, the match produces an environment that gives the values for all bound variables. For example, matching the formula, "event1[attr →A]", against the event "event1[attr=3]" produces an environment that binds "A" to 3.

6.4.3 And

We match $f \wedge g$ to an event by first matching f and then g to the event. If f (g) reduces to true, its environment becomes part of the environment of the parent *And* and is therefore available when matching g (f). If both (either) of f and g reduce to true (false), $f \wedge g$ reduces to true (false). Otherwise, $f \wedge g$ reduces to the $\wedge$ of the reduced f and g. If and when $f \wedge g$ reduces to true, its environment contains the bindings for all variables that f and g bind. It is an error to bind the same variable multiple times.

6.4.4 Or

We match $f \vee g$ to an event by first matching f and then g to the event. If f (g) reduces to true, $f \vee g$ reduces to true with the environment from f (g). If neither f nor g reduce to true, $f \vee g$ reduces to the $\vee$ of the reduced f and g. Because $\vee$ is short-circuit (i.e., it stops as soon as f or g reduce to true) the $\vee$ will end up with the environment from either f or from g. Consequently, $f \vee g$ may bind different variables depending on which of f or g reduces to true first.

6.4.5 Not

We match $\neg f$ to an event by matching f to the event. If f reduces to false (true), $\neg f$ reduces to true (false). Otherwise, $\neg f$ reduces to the $\neg$ of the reduced f. $\neg f$ never produces a variable environment.

6.4.6 Basic Next

Recall that "$\bigcirc f$" matches a subtrace $(e_i e_{i+1} \cdots e_j)$ if f matches $(e_{i+1} \cdots e_j)$. Thus, when we first match an instance of "$\bigcirc f$" to an event, "$\bigcirc f$" reduces to f. We then match f against subsequent events; if f reduces to true, it produces an environment with the variables that f binds.

6.4.7 Eventually

Every time we match an instance of $\Diamond f$ to an event, we create a new instance of f. Then we match all instances of f (including ones created on earlier match attempts of $\Diamond f$) in order of creation to the event. As soon as one of the f instances reduces to true, the *Eventually* immediately reduces to true. Consequently, on a reduction to true, *Eventually* gets the environment from the f that reduced to true. *Eventually* reduces to false only if we reach the end of the trace and none of the f instances evaluated to true. Since this can be expensive for long traces, users can optionally specify a limit (similar to Brzoska's metric [73]). If an *Eventually* with limit L remains pending even after seeing L events, it reduces to false.

For example, consider matching "$\Diamond(\bigcirc$F)" against the trace "E E F". When we match the *Eventually* against the first E, we create an instance of "$\bigcirc$F" and apply it to the first E. Since this is the first event to which we are applying this instance of "$\bigcirc$F", it reduces to a pending F. When we match the *Eventually* against the second E, we create another instance of "$\bigcirc$F" and attempt to match the two instances to the second "E". The first instance reduces to false (because E $\neq$ F) while the second reduces to a pending F. Finally, when we match the *Eventually* against the F, we create yet another instance (so now we have one false instance of "$\bigcirc$F", one pending instance of F, and one pending instance of "$\bigcirc$F"). We match the two pending instances against F and the first one reduces to true and thus the *Eventually* reduces to true.

6.4.8 Until

Every time we match $f \ \mathcal{U} \ g$ to an event, we create a new instance of g *as long as* f has matched all previous events or if there were no previous events. If any of the g instances reduces to true, *Until* reduces to true with

the environment from the first g that reduces to true. If all of the g instances reduce to false, the *Until* reduces to false.

For example, consider matching "E $\mathcal{U}$ (○G)" against the trace "E F G". When we match the *Until* against the E, we create a new instance of "○G" and match it against the E; the "○G" instance reduces to a pending "G". When we match the *Until* against F, we again create another instance of "○G" because the previous events so far (i.e., just E) have all matched the formula E. The first instance (which is a pending "G") reduces to false. The second instance (a newly created "○G") reduces to a pending "G". When we match the *Until* against G, this time we do not create a new instance of "○G" because the previous event was not E. However, when we apply the pending instance of the "○G", it reduces to true and thus the *Until* reduces to true.

6.4.9 Producing useful output from the matches

As described so far, matching a formula to a trace produces an environment. This section describe how we extract information from an environment.

Simple queries

To extract output in a form that is useful to the user, we wrap a formula into a *query*:

$$x_1, \cdots, x_n : f$$

A query associates a sequence of expressions $(x_1, \cdots, x_n)$ with a formula (f). For each successful match of the formula, we produce a row in a CSV output file which contains one column for each expression.[4] For example, consider our example from Section 6.2:

```
R, T1, T2, T3, TotalTime = T3 - T1:
 RequestReceived[Timestamp →T1, Id →R]
  ∧ ○[Id = R] ( GetBalance[Timestamp →T2, Id = R]
   ∧ ○[Id = R] (ResponseSent[Timestamp →T3, Id = R]))
```

For each successful match of the above formula in a trace, we produce a row with five columns: for R, T1, T2, T3, and TotalTime (which is T3 - T1).

[4]The $\mathcal{U}^*$ extension extends this slightly (described below)

Queries with loops

The above queries extract the values of bound variables from a fixed number of events. This section introduces two extensions that effectively add a loop to the *Until* operator.

First, we observe that the until operator ($\mathcal{U}$) stops matching as soon as it reduces to true. Sometimes we need to find *all* occurrences. For example, let's suppose a process, as part of its work, needs to start many threads and we suspect that there may be undue delay between the start of the process and the start of one of the threads. The following pattern finds the delay for the first thread:

Pid, Tid, delay = T2-T1 :
StartProcess[Id →Pid, Timestamp →T1] ∧
(¬EndProcess[Id=Pid]
$\mathcal{U}$ StartThread[Id = Pid, TaskId →Tid, Timestamp →T2]

To find the delay for all threads, we replace $\mathcal{U}$ with $\mathcal{U}^*$. $\mathcal{U}^*$ behaves the same as $\mathcal{U}$ except that it captures the current continuation every time we match its instance to an event. If the match reduces to true, we propagate that (as for the normal $\mathcal{U}$) to the parent formula *and* resume the continuation starting with the next event (i.e., the continuation pretends that $\mathcal{U}^*$ did not reduce to true and continues matching at the next event). In the context of our example, while $\mathcal{U}$ would have produced only a single output row (for one *Tid*), $\mathcal{U}^*$ produces many rows, one for each distinct *Tid* that the process starts.

Second, while $\mathcal{U}^*$ outputs one row for each match, sometimes we want to aggregate all the matches. We can easily do this by processing the output from $\mathcal{U}^*$; however, since this is a common idiom we directly support it. For example, let's suppose that an operation acquires and releases locks multiple times during its execution and we want to determine the total lock holding time across all the acquisitions. For this usage we support a *Query-Until* operator.

A Query-Until specifies a subquery that repeatedly matches until the "Until" formula matches. Specifically, $\{Q\}$ $\mathcal{U}$ U is a query expression such that: (i) Q binds a single variable with the value of an aggregation expression (e.g., *Sum*, *Min*, *Max*, or *Mean*)); (ii) we evaluate Q on every event while updating the aggregated variable until U matches. When U matches, the *Query-Until* reduces to true with its environment containing only the variable that Q binds. E.g.,

LockHoldingTime:
RequestReceived[Id →R] ∧

({LockHoldingTime = Sum(T2 - T1) :
AcquireLock[Timestamp →T1, Id = R] ∧
(◇ReleaseLock[Timestamp →T2, Id = R])}
$\mathcal{U}$ (○ResponseSent[Id = R]))

The query part of the *Query-Until* accumulates the lock holding times in the variable *LockHoldingTime* until the next event is a *ResponseSent*. On a match of the query, *LockHoldingTime* contains the total lock holding time from any number of lock acquisitions and releases.

6.5 Implementation considerations

Since the ◇ and $\mathcal{U}$ create new instances of formulas at each event, our naive implementation could not analyze many of our larger traces: with each new event in the trace we would end up with additional formula instances which would have to be matched against subsequent events.

To avoid this cost, we maintain maps from combinations of attribute values to the formula instances that are *waiting* for those values. Thus, for example, if our formula is:

$\cdots$ ○[A = a] $\cdots$ ○[A = x, B = y] $\cdots$

we would have two maps, one for the attribute "A" and one for the pair of attributes "(A, B)". The first one maps values of "A" to formulas waiting on that value; the second one maps values of the pair "(A, B)" to the formulas waiting on that value. When we encounter an event, we look at its set of attributes and find all the maps that may be relevant to the event. For example, if our event is *E[Timestamp=1, A=10, B=20]*, both the "A" and the "(A, B)" maps are relevant. We then use the attribute values in the event to look up the maps and thus readily find the formula instances that are possibly waiting for our event.

6.6 Results

TPA originated from a real need: when debugging performance problems at Google we had to reason over many different kinds of traces (kernel, file system, etc.). Reasoning over these traces was slow, error prone, and laborious. We would use standard Unix tools to search for relevant events in the traces and manually reason over the chain of events. In some cases we wrote Python code to implement formulas but that was inflexible: developing formulas often requires exploration and iterative refinement which was inconvenient in Python because the formulas were one or two orders of

magnitude larger than their TPA counterparts. TPA enabled us to develop formulas iteratively and intuitively. We now evaluate TPA with respect to generality (Section 6.6.2), scalability (Sections 6.6.3 and 6.6.4), and usefulness (Section 6.6.5).

6.6.1 Methodology

We evaluated TPA on real production traces from computers in Google datacenters; these computers concurrently run multiple services and each service concurrently handles tens to thousands of user requests per second. We analyzed these traces on a desktop workstation (with four Intel Xeon cores running at 2.7 GHz and with 8 GB of memory).

TPA is implemented in about 5000 lines of Java code. This code parses formulas from an input file and matches these formulas against an input trace in a single pass. At each event in the trace, TPA instantiates formulas as needed and matches the instantiated formulas against the event. The implementation directly follows from the descriptions in Sections 6.4 and 6.5.

6.6.2 Generality

TPA analyzes traces that are in a source-agnostic binary format. We have straightforward converters from many different kinds of traces to this format; thus we can readily use TPA to analyze traces from diverse sources. This section describes our experience in analyzing traces from three different sources: (i) kernel traces record every transition in and out of the kernel along with high level events that enable us to tie kernel events to RPCs; (ii) user request logs which contain events at each stage of processing a user request to GMail; and (iii) file system traces record the start and end of each file system and disk operation.

Since kernel traces generate a million or more events per second, a naive implementation could perturb the underlying system. Google's implementation is carefully crafted to collect these traces in an in-memory buffer with less than a 3% overhead. The size of the buffer limits the size of the trace that we can collect; at the default size, we can collect about 20 seconds before the buffer fills up. If we are exploring rare events, we may collect 20 second traces from many computers and analyze all of them with TPA to make sure we get enough instances of our long-tail event.

The difficulty and labor involved in manually analyzing these traces makes them ideal candidates for TPA: once an expert specifies the formulas, TPA can do the hard work of matching the formulas against traces. These formulas encode persistent knowledge about particular performance phenomena. For example, let's suppose we (the experts) develop formulas

to quantify a particular performance problem observable in kernel traces from machines running GMail. Others can use these formulas to check for the same problem in their systems and we can use these formulas over time to make sure that our performance problems do not recur.

We developed formulas for our traces while tracking specific performance problems in Google services. All formulas, except for one, use two or more temporal operators ($\bigcirc$, $\Diamond$, or $\mathcal{U}$); thus, if we did not have TPA we would have had to manually implement state machines to match these sequences. We now describe the process of developing one of our simplest and one of our most complex formulas.

The first example calculates the type, size, and duration of each file system operation:

Duration=T2-T1, Op, Bytes :
fs_enter[Timestamp →T1, op →Op, pid →Pid]
 ∧◇fs_exit[Timestamp →T2, ret →Bytes, pid=Pid]

fs_enter marks the start of a file-system request and fs_exit marks its end. Since the enter and exit must be in the same process and multiple of these requests cannot be pending from the same process, this formula uses the process id (Pid) to associate the fs_exit with its fs_use. The return value of the fs_exit gives the number of bytes that the operation read or wrote.

The second formula, calculates the duration between an RPC request arriving on a socket and the end of the RPC. The RPC mechanism works as follows (Figure 6.1): (a) the *epoll_wait* detects that some data has arrived for one or more RPCs; (b) *recvfrom* system call reads the data; (c) *start_rpc* marks the starting of RPC processing by the application; (d) *sendto* system call sends the response for the RPC; and (e) *end_rpc* marks the end of the RPC in user space. Existing Google tools measure and quantify the duration between *start_rpc* and *end_rpc* (i.e., the "Application duration"): these timestamps are easy to gather and analyze because they are the two ends of application-specific RPC processing. However, this duration ignores many of the costs of the RPC, namely the time between the *epoll_wait* and the *start_rpc*. We do not yet have a way of directly measuring "Actual duration" because (i) part of the RPC processing happens in the kernel, part in an RPC library, and part in the user application; (ii) at the time of the *epoll_wait* and *recvfrom* we do not know what RPC we are processing: it is only after we have looked at the received data we know the identity of the RPC. Thus, the pattern we are about to describe sheds light onto a delay that we have no other way of measuring at Google.

When we start writing this pattern, we run into the same issue discussed above: with many RPCs arriving per second on a computer, how do we know that a given *recvfrom* is for our RPC or if it is for some unrelated

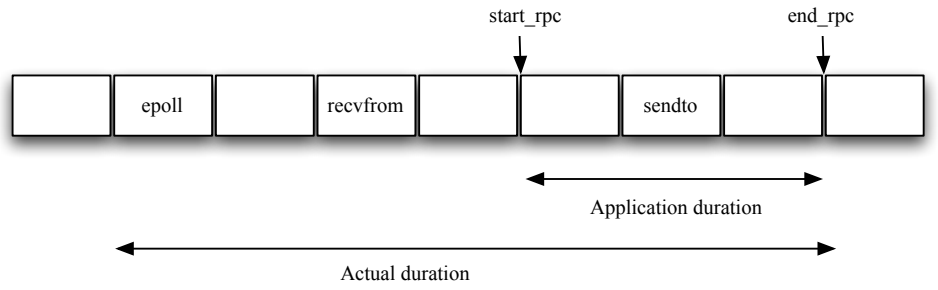

Figure 6.1: Events involved in RPC

RPC? Our main insight for addressing this is to recognize that the socket that *sendto* writes to is the same socket that *recvfrom* must have read from (this is a property of Google's RPC implementation). Fortunately, our kernel traces record the socket for the *recvfrom* and *sendto* in the *arg1* attribute of the event. Thus we arrive at the following formula:

syscall[name="recvfrom", arg1 →Sid]
 ∧(○(¬syscall[name="recvfrom", arg1=Sid]
  $\mathcal{U}$ (start_rpc[RpcId →Rid] ∧
  ◇syscall[name="sendto", RpcId=Rid, arg1=Sid])))

To extend our formula to also include the *epoll_wait* we assume that the latest *epoll_wait* before the *recvfrom* is the one that notices that data has arrived for the RPC (this is an optimistic assumption though likely to be true given our knowledge of the implementation). We get the formula:

syscall[name="epoll_wait", ret > 0, Timestamp →T1]
 ∧(○(¬syscall[name="epoll_wait", ret>0]
  $\mathcal{U}$ (syscall[name="recvfrom", arg1 →Sid]
   ∧(○(¬syscall[name="recvfrom", arg1=Sid]
    $\mathcal{U}$ (start_rpc[RpcId →Rid, Timestamp →T3]
     ∧◇syscall[name="sendto", RpcId=Rid, arg1=Sid])))))

The *epoll_wait* returns a value greater than zero if it finds data waiting on a socket; thus our formula looks specifically for such an *epoll_wait*. Our formula handles the situation where an RPC with a large request requires multiple calls to *recvfrom*.

Using the environment for a match of this formula, we can readily calculate "Total Time". Finally, we notice an additional complexity: a given *epoll_wait* call may notice that there is data not only on one socket but on many sockets; thus a given *epoll_wait* may be part of the match

for many different RPCs. We handle this using a $\mathcal{U}^*$ instead of $\mathcal{U}$ for "$\mathcal{U}$ (syscall[name="recvfrom", arg1 →Sid]".

In summary, we have found that TPA is useful for analyzing at least three kinds of traces. Given an in-depth knowledge of a trace's events, TPA formulas follow intuitively and directly from this knowledge.

6.6.3 Scalability: time

Figure 6.2 shows how the TPA scales with long traces. Since the kernel traces were our longest traces and had the most complex formulas, Figure 6.2 shows the performance of four of formulas a one billion event kernel trace: (i) w computes delay while RPC processing is blocked because a thread is not available; (ii) b computes time spent blocked on a lock; (iii) c computes time spent executing on a CPU (i.e., doing real work); and (iv) e is the epoll_wait pattern from Section 6.6.2. The performance of our other formulas lies between w and e and thus we omit them to avoid clutter in the graph. A point (x, y) with symbol f says that TPA processed y events per second for formula f when TPA was x events into the trace; thus lower means slower. To increase the generality of our results we evaluated the scalability of each formula on two additional traces of length 6M events and got results comparable to those in Figure 6.2.

From Figure 6.2 we observe that the number of events processed per second for each formula remains largely and surprisingly constant. Even though it is easy to produce a formula that does not scale to a large trace (e.g., a formula with an *Eventually* that never reduces to true will require as many instances as the number of events in the trace) this does not happen in practice: most *Eventually* in correct formulas reduce to true in a small number of events and thus the pathological situation does not occur. To guard against poor scalability, users have the option of specifying limits for temporal operators (Section 6.4.7)

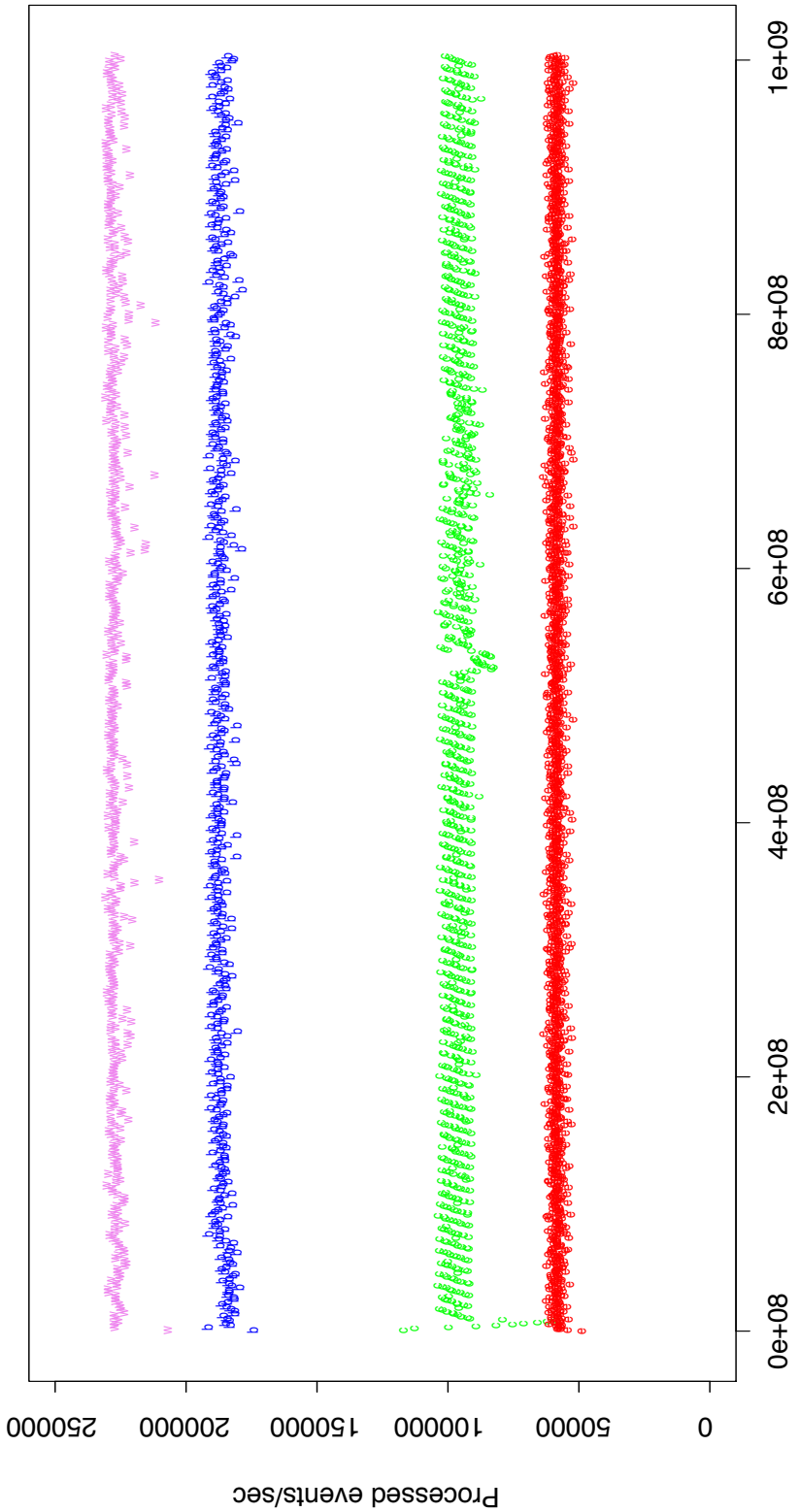

Figure 6.2: Scaling behavior for long traces.

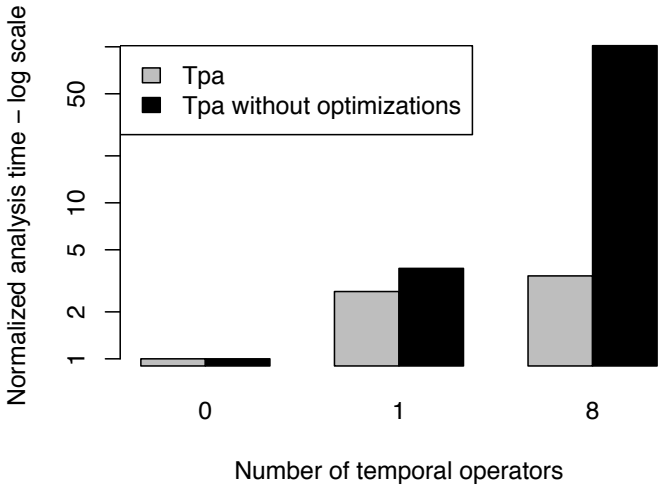

Figure 6.3: Benefit of the optimizations in Section 6.5. Analysis times normalized to left-most bar in graph.

Figure 6.2 shows the scaling behavior for individual formulas; however, TPA can match multiple formulas in a single traversal of the trace. For a 9 million event trace, matching 7 formulas (including the ones in Figure 6.2) back to back takes 725 seconds; matching them all in a single trace traversal takes 399 seconds.

Figure 6.3 compares the analysis time of TPA with and without the technique in Section 6.5 on three formulas: (i) 0: a formula that matches a single event; (ii) 1: a formula with one temporal operator; and (iii) 8: a formula with 8 temporal operators (the epoll_wait formula from Section 6.6.2). We see that as are formulas become more complex (i.e., have more temporal operators) our optimization becomes increasingly important.

In summary, TPA scales at least to traces that are 1B events.

6.6.4 Scalability: memory

By default, TPA runs with a maximum Java heap size of 2 Gb on a 32-bit Java JVM. To see if TPA could run in less memory, we analyzed a 9 M event kernel trace repeatedly using the formulas in Figure 6.2. For each experiment we halved the size of the maximum Java heap (using the "-Xmx" flag) to find the size at which the analysis time degrades.

We found that the analysis time was unchanged for heap sizes down to 128 MB but degraded significantly for 64 MB. Thus even the cheapest laptop or desktop computer today has enough memory to analyze traces that are millions of events.

6.6.5 Usefulness

The goal of TPA is to help in understanding performance; in some cases, this understanding may immediately enable us to *improve* performance while in other cases it may simply point us to components that are worth investigating. The following case studies illustrate the range of performance problems for which we have used TPA.

Scenario 1: How rare is an event?

Performance anomalies are everywhere. Indeed, every time we visualize a new trace our first reaction is usually: "what is this odd behavior?". Even the experts in the system that produced the trace are often surprised by the gap between what they expect to see and what actually happens.

While we could attempt to track down every anomaly that we see, it is usually not worthwhile to do so: many anomalies are rare and thus time spent tracking them down will yield little benefit. Thus, we need a way to separate the anomalies that are frequent and must be studied from anomalies that are too rare to be worth an investigation. We have frequently used TPA for such analysis.

The example in Section 6.3.5 illustrates such a use of the TPA. While looking at data from a stress test we encountered a situation where there was significant contention on the spin lock for the waiting queue. This situation was believed to be extremely rare and thus not worth optimizing. That we saw an instance of it in a trace from a stress test was perhaps not too surprising. Thus, to see if this actually happened in practice we wrote the formula in Section 6.3.5 and applied it to 4 traces taken from real production load (i.e., not stress test). We found that this situation happened between 2 and 10 times per second per computer (each trace from a single computer) and each time it occurred it delayed the operation by several milliseconds.

Thus, TPA enabled us to quantify this problem as rare (about 0.1% of the lock acquisitions and releases suffer from it) but still more than what the designers expected. Without TPA we would have had to resort to searching and manual labor; for example there were more than 4000 calls to *nanosleep* in one trace and only 8 of them were relevant to our situation.

Scenario 2: Where does all the time go?

There are many possible situations that contribute to long tail latency for an RPC: the RPC may be doing too much work, it may be waiting on a lock, it may be waiting for a CPU, it may be waiting in a queue behind other RPCs, it may be waiting on a disk access, etc. For a single RPC we can, with some

manual labor using an appropriately powerful visualization, determine the breakdown to account for 100% of the time for an RPC. However, we really want to know this breakdown for not a single RPC but a large number of RPCs so that we can make informed decisions on what to optimize next.

To do this, we can write formulas; one for each component of a breakdown. In addition, we can write a formula that quantifies the total latency of each RPC. We can now use TPA with these formulas to determine if the components largely account for the total latency; if they do not, then we know that we have not accounted for all the components of latency and must write new formulas.

We wrote eight formulas to break the time spent in a no-SQL server [74] RPC and found that three of the formulas dominated. The three that dominated were: Cpu (i.e., time doing actual work on behalf of the RPC accounted for nearly 50% of the latency), Waiting for Thread (i.e., when the RPC work is on a queue waiting for a thread to pull it off and work on it accounted for about 30% of the latency), and Blocked (i.e., waiting to acquire a lock or waiting for a callee RPC to finish accounted for about 20% of the latency). On average, doing actual work actually consumed less than half of the overall time for the RPC. Moreover, the distribution was skewed: for example, *Waiting for Thread* was bi-modal: it was tiny for most RPCs but was the dominant contributor of latency for some RPCs.

Thus, TPA revealed a possible cause of long-tail latency in the no-SQL system; this and other follow-up investigations led to changes in the threading model for the system.

Scenario 3: What is the true cost of an operation?

Concurrent code often uses thread pools [75]. Intuitively, a service submits tasks to the thread pool; if the pool has an available thread, it wakes up the thread and gives it the task to work on. If it does not have an available thread, it may create a new thread or queue up the task for the next thread that becomes available. If we over-provision the number of threads, the thread pool always has an available sleeping thread for each incoming task; if we under-provision the number of threads, most incoming tasks will have to wait on a queue before a thread picks it up.

While profiling a component of GMail, we noticed that threads going to sleep (while waiting for work) and threads waking up (when a new task arrives) was contributing significantly to the latency. The code that accomplishes this uses the *futex* system call to both go to sleep and to wake up a sleeping thread. Unfortunately it is not easy to measure the cost of the two uses of *futex* for two reasons: (i) when a thread uses *futex* to wait for work, the operating system will often context switch to another thread; thus, we need to differentiate the time spent in *futex* from the time the thread was

context-switched out; (ii) the two uses of *futex* are distinguishable by the sequence of events: the thread that waits for work will have a context switch during the futex call while the thread that wakes up another thread will have a *sched_wakeup* event during the *futex* call.

Fortunately both of these sequences are easy to recognize using formulas in TPA. This study resulted in an effort to tune the thread pool for GMail and to improve the cost of putting a thread to sleep and waking it up. These efforts have already yielded a 5% improvement in latency for one component of GMail.

6.7 Related work

There is much prior work on extracting rich information from program runs (e.g., DTrace or AOP [68, 69]); however, most of these approaches require us to add instrumentation to programs. For example, DTrace inserts probes and actions into the system under investigation. This is not feasible in our production setting: the time to add new instrumentation, get it reviewed and deployed, and finally to collect data from it takes weeks and is thus inconvenient for exploration. Moreover, this instrumentation may itself perturb the system. Therefore it is critical to mine information we already collect whenever possible.

6.7.1 Temporal logic

The main contribution of this work is not a new language but a novel use of an existing language: LTL. Thus, our contributions are in how we use and implement the language rather than a language design.

Unlike TPA which focuses on extracting performance information from traces, prior work (with the exception of Finkbeiner et al. [76]) focuses on system verification [71, 72, 77–85] or debugging [86]. Thus, none of these prior systems add variables to LTL which are crucial for extracting performance data from traces.

Finkbeiner et al. [76] is the closest work to this work: they also note that one could use LTL to analyze program traces to compute "experiments" (properties at particular events in the trace) and "aggregate statistics". This is a theoretical paper and the concepts are similar to many of the concepts in our work. However, unlike our work, they do not introduce a notion of variable bindings (which are key to reasoning over sequences of events), describe optimizations necessary for scaling to realistic traces, or apply LTL to solving real world performance problems.

Prior work uses two broad techniques for reasoning over formulas in temporal logic. The tableau method is useful for determining whether or

not a formula is satisfiable [87]; in contrast, we wish to determine if a formula matches a particular trace. The automata method converts LTL formulas into automata and uses it to extract information from traces (e.g., Finkbeiner et al. use an algebraic alternating automata [76]). TPA's implementation strategy is similar to using an automata except that because of the use of variables we also need to maintain environments as we match formulas against traces.

6.7.2 Trace query languages

We use our extended LTL as a domain-specific language for analyzing performance traces. Thus, while there is much prior work in analyzing traces none of the prior work focuses on performance traces; thus the idioms they support are inconvenient for our needs.

Many systems in the Complex Event Processing community (CEP) describe event matching languages [88–92]). Unlike our work, which focuses on analyzing performance traces, these papers focus on RFID-based inventory management, financial services, click stream analysis, and health systems. Consequently, our language is intended to be convenient at expressing idioms in performance traces (e.g., by supporting the constrained *Next* operator). CEP languages decouple the event sequence matching and the attribute matching by using the following construct: *EVENT <sequence>WHERE <attribute matching>*. Our performance traces require a lot of attribute matching (e.g. we need to match a lock acquisition with a lock release from the same thread). By using variable binding and describing both event and attributes match at the same time, our formulas are significantly more compact for our needs. In addition, unlike the CEP literature that introduces many new operators, we preferred to restrict ourselves as much as possible to a well-known and understood notation: LTL. In this way we hope to end up with a notation that is easy to understand and reason with.

PQL [93] and PTQL [94] use an SQL-like syntax to perform queries on a event trace. PQL targets finding application errors and security flaws, while PTQL targets debugging and profiling. Both systems target Java programs and their events are high-level Java events such as method invocations, allocations, and field accesses. While SQL notation is familiar to programmers, it is not convenient for expressing complex temporal interactions between events. For example, PTQL cannot conveniently express simple sequences such as "Thread A invokes M, and next Thread A invokes B or C" because there is no convenient way of saying: between these two events in thread A there must be no other events for thread A.

KOJAK [95] uses the same parallelism to analyze traces as the original run that produced the traces. Unlike TPA, they use a general-purpose

language for matching patterns against traces. While general-purpose languages can express any pattern that TPA can express, their implementations of patterns are one to two orders of magnitude larger (in our experience) than in TPA and consequently less convenient for exploration.

6.7.3 Other related work

Altman et al. [96] show that for large-scale applications, performance problems often emerge as idle time. Their methodology to identify the root cause of idleness in programs uses sampling of the execution state of a Java Virtual Machine. By inspecting the stack they classify the idle time into categories such as locking contention, garbage collection, and resource constraints. One of the usage scenarios for TPA is also as an "idle-time classifier" (Section 6.6.5). However, unlike our work that operates on fine-grained traces, Altman et al. is limited to sampling in Java Virtual Machines.

Xu et al. [97] use data mining techniques to detect performance anomalies using the applications' source code and console logs. The purpose of this tool is to automatically detect performance anomalies by reading the log file one line at a time. This approach does not detect complex interactions that require reasoning across several log entries.

6.8 Conclusion

For large-scale applications the slowest 1% operations (i.e., the long-tail latency) cannot be ignore as "outliers". The latency of the slowest operations affects capacity planning (and thus costs) and for services that serve hundreds of millions of users, there are, by definition, many user requests daily that are in the slowest 1%.

To analyze long-tail latency, we need to analyze traces. However, analyzing traces is difficult and laborious: one must reason through long sequences of events to find the cause of a performance problem. This chapter describes and evaluates a system, TPA, that, given domain knowledge encoded as formulas, can analyze large traces. Thus, TPA can alleviate much of the manual effort in analyzing traces.

We show that TPA scales to large traces: even for our most complex formula, TPA can analyze more than 50000 events per second using only 128M of memory. We demonstrate the usefulness of TPA by presenting three case studies, two of which have directly contributed to improving the performance of Google services.

Chapter 7

Conclusion

This dissertation investigated several challenges regarding commercial computing platforms and real-life workloads. In this chapter, we first summarize these challenges and then detail the conclusions that can be drawn from this research work. In addition, we highlight interesting research topics that could be investigated further in the future.

7.1 Summary

The rise of cloud computing, where most storage and compute power is located in the datacenter, comes with a number of challenges and opportunities. A major challenge is to provide a satisfying experience to the user. This is obviously closely coupled with the performance of the application in the datacenter. Studies have shown that users do not tolerate long latencies and leave the website, which leads to user drop-out and means a loss in users and thus a loss in revenue. For complex applications, we typically see a latency distribution with a long tail: the performance of most requests is located around the average latency, but there is a small percentage of long latencies, for example the slowest 1% of requests takes 10 times longer than the average request. While profilers and hardware simulators can help with improving the average latency by measuring the performance of the software and testing the performance on future hardware platforms, they are of little help for the long tail latencies. Only a small number of requests suffer from these problems that are often caused by complex interactions in various layers of the execution stack spanning both software and hardware.

Besides these challenges there are also major opportunities in cloud computing. One of the major costs in datacenters is power usage. This includes the power consumed by the IT infrastructure (e.g., servers, network devices, etc.) and power for cooling and power transformations. Where datacenters used to be power-hungry buildings with large cooling installa-

tions, companies are nowadays pushing towards free cooling by designing the building from the ground up and building in areas with a favorable climate. Large-scale Web applications like Google, Amazon or Facebook build their own custom-designed datacenters with the purpose of lowering the cost. In this work we focus on the energy efficiency of the computing hardware, as this is a the core of the power usage: lower power used by the IT infrastructure means less cooling and less power transformations. Thus optimizing the energy efficiency of the server has a very large impact on the energy efficiency of the whole application and datacenter.

In the following subsections, we briefly highlight the major findings and contributions of this work to the performance and energy efficiency of commercial computing platforms.

7.1.1 Architectural simulation

Simulating future hardware platforms using an architectural simulator gives us insight in the performance of the application on a hardware platform before building it. Various tradeoffs can be researched: such as the performance versus the energy efficiency of an application on that specific hardware platform.

Architectural simulators come in different flavors, as there is a tradeoff between speed and accuracy. Cycle-accurate simulators model the hardware in a great level of detail and simulate the behavior of each component at every cycle. This makes them the most accurate simulators but also very slow. Complex applications with large amounts of data can take a long time to reach a phase of stable performance. In particular, it can take hours before a software cache is properly warmed up. Cycle-accurate simulation cannot be used to simulate the workload as a whole, as it is simply too slow.

Interval simulation is a technique that abstracts away most hardware components into an analytical model, making it faster but less accurate than cycle-accurate simulation. Prior work in interval simulation was validated against a cycle-accurate simulator, but not against real hardware. In this work, we validate interval simulation against real hardware, namely an AMD Opteron 2350 quad-core server processor. The average error compared to real hardware is 9.8% for the micro-benchmarks, and 18.6% for a set of CPU-intensive benchmarks including SPECjbb2005, H.264 video decoding and encoding, bio-informatics, and multi-threaded PARSEC benchmarks. We obtain similarly accurate results for a non-trivial Web 2.0 search engine server workload: 7.0% average error for response time and 12.7% for throughput across a range of concurrent clients. We also study the trade-off in speed versus accuracy when enabling sampled simulation. The end result is a full-system software simulator that faithfully simulates x86 hardware at a speed in the tens of MIPS range: one particular sampling

strategy achieves a speed of 37 MIPS and an average error of 23.1% for a set of CPU-intensive workloads.

VSim takes the speed versus accuracy tradeoff one step further: VSim models the target processor as a fixed performance ratio to the host processor. This is clearly less accurate than interval simulation, but enables very fast multi-node simulation. VSim is a novel full-system simulation methodology that leverages virtualization technology to simulate multi-server setups. VSim consists of a system virtual machine that runs on a host server and controls CPU, network and disk performance as perceived by software, i.e., the software is given the illusion to run on a target system with performance properties that differ (significantly) from the host but mimic the target system. Virtualization also enables simulating multiple target servers per host by running target servers as guest virtual machines. Distributed simulation across multiple hosts enables simulation at scale. The implementation of VSim in VirtualBox and the evaluation presented in Chapter 3 illustrate its accuracy: 2.0%, 4.4% and 4.9% average error against the modeled CPU, disk and network performance, respectively; complete workloads (Lucene and Olio) involving CPU, disk and network activity are shown to be accurately modeled in VSim (average error of 3.2%). These results are obtained at a simulation slowdown of one order of magnitude only compared to native hardware speed, and our current implementation can simulate up to five target servers per host.

7.1.2 Energy efficiency and proportionality

Recently, SPEC launched SPECpower, a benchmark for evaluating the power and performance characteristics of computer servers [10]. Rivoire et al. [11] propose JouleSort, a sort benchmark aimed at evaluating the energy efficiency of a wide range of computer systems from servers to embedded systems.

Using the results provided by SPEC for SPECpower, we analyze how energy-proportionality has evolved over the past three years on a broad set of contemporary servers. We evaluate how well the SPECpower score quantifies energy-proportionality and how much energy can be saved by making servers more energy-proportional. We conclude that energy proportionality has improved significantly over the past few years, from 30 to 40 percent in 2007 to 50 to 80 percent in 2011. Yet, substantial energy savings might be possible to achieve by further improving a server's energy proportionality. Closing the gap between today's most energy-proportional system and the ideal energy-proportional system could potentially lead to an energy (and proportional cost) saving of 34 percent.

Although SPECpower and alike benchmarks offer valuable insight in the energy efficiency of a computer system, they have limited flexibility.

The benchmarks are rigid and cannot be altered to reflect different workload behaviors. In particular, EEMBC's EnergyBench is tied to the EEMBC performance benchmarks; the SPEC power benchmark is a Java server workload that generates and completes a mix of transactions; JouleSort implements a sort algorithm. These benchmarks are unable to explore the energy efficiency of computer systems across the workload space. In other words, the numbers produced by these approaches may be limited in scope (they are tied to these specific workloads) and it is hard to generalize towards other types of workloads, i.e., a computer system that is energy-efficient for the power benchmark does not necessarily imply that it is energy-efficient for other workloads.

We therefore propose SWEEP, a framework for generating synthetic workloads with specific workload characteristics in order to generate compute-intensive workloads, memory-intensive workloads, I/O-intensive workloads, and any mix thereof. In particular, SWEEP enables its users to configure the workload's characteristics by setting the ratio of integer versus floating-point instructions, the inter-instruction dependencies, memory access patterns, disk I/O access patterns, etc. SWEEP provides a unique opportunity to its users: it allows for exploring the energy efficiency and performance of computer systems by 'sweeping' across the workload space.

Using SWEEP we generate a range of synthetic workloads with very different characteristics and run these workloads on two real hardware systems, a low-end system (Intel Atom) as well as a high-end system (AMD Quad-Core Opteron), and evaluate their energy efficiency across different workload behaviors. We conclude that whether one machine is more energy-efficient than another machine is very much workload dependent.

To visualize a machine's energy efficiency relative to a reference machine, we propose the Energy-Delay Diagram (EDD). The EDD represents the trade-off in performance versus energy in a more intuitive way than the traditionally used EDP and ED^2P metrics do.

7.1.3 Long-tail latency analysis

Long-tail latencies are often caused by complex interactions between various layers of the execution stack, and not in the least part by contention on resources (e.g., locks). To analyze long-tail latency we must reason over fine-grained traces (i.e., timestamped sequence of events) which is laborious and requires extensive domain knowledge. Reasoning with traces is difficult because we must reason across chains of events where each event affects subsequent events. For example, let's suppose we wish to find how often a high-priority thread, e.g., H, waits for a lock that a low-priority thread, e.g., L, holds (i.e., this is a case of priority inversion). Given a ker-

nel trace, we can readily find when H waits on a lock (it shows up as system calls to *futex* in the Linux kernel trace). To determine if H was waiting for a lock held by L we must now reason across time to determine if L released the lock just before H acquired it. Using manual effort and search tools we can do this reasoning for a few operations only. However, reasoning over a few operations does not tell us if the priority-inversion is common or rare. Thus, we must do this reasoning over many long traces.

We therefore propose TPA, a language based on linear-temporal logic extended with variables to analyze traces and thus reduce the manual labor involved in analyzing traces. We show that TPA scales to large traces: even for our most complex formula, TPA can analyze more than 50,000 events per second using only 128 MB of memory. We demonstrate the usefulness of TPA by presenting three case studies, two of which have directly contributed to improving the performance of Google services.

7.2 Future work

To conclude this dissertation we elaborate on some future research directions in hardware simulation and analysis of performance on production systems.

7.2.1 Hardware simulation

Chapter 3 introduced VSim, a novel simulation technique based on hardware-assisted virtualization. The major limitation of our current simulation approach is the CPU model: the performance of the guest CPU is modeled as a fixed factor of the host platforms CPU. In reality, however, the performance factor is defined by the workload and varies over time during the execution. Depending on the workload's characteristics and phase behavior, the performance factor might change in time. For this work we calculated a fixed performance factor for each workload.

Multiple strategies could be developed to cope with this problem. For one, we could use performance counters to measure the workloads characteristics while running the simulation. These characteristics could be used to determine the performance factor of the next simulation quantum. Synthetic benchmarks could be used to automatically generate the CPU model. Running the synthetic benchmarks on both the target and host system gives us performance factors for a wide range of applications, from which we can generate a CPU model.

On the other hand, we could use a detailed simulator (this could be a cycle-accurate or interval simulator) to determine the performance model. By checkpointing the architectural state in VSim and transferring that state

to the simulator, we would be able to determine the performance factor for that workload. We could then use VSim to simulate the workload faster and at a larger scale. This way we combine the accuracy of a detailed simulator with the speed and scalability of VSim.

A second hurdle in simulating multiple targets on one host is the memory usage of the simulator. The naive approach is allocating all memory required for each target on the host. This requires a host with a tremendous amount of memory and yields a very expensive simulation machine. Modern hypervisors have techniques to share pages with the same content across different virtual machines. VSim is based on VirtualBox, the page sharing feature is implemented in a newer version of the hypervisor. An interesting avenue for future work is to port VSim to the latest version of the hypervisor and analyze the reduction of memory usage that can be achieved by using same page sharing. The benefits of this technique will obviously vary a lot across different benchmarks. One can imagine that this yields good results when the targets are all running the same workload on the same input data, for instance a cluster of Web servers serving the same application. For map-reduce like workloads, where a set of data is distributed across a set of nodes, the results will probably be less impressive. Further research could give us more insight in the behavior of these techniques and the impact on the simulation of clusters.

7.2.2 Energy efficiency and proportionality

Chapter 4 showed that there is a trend to more energy-proportional computers. Especially the CPU has come a long way, thanks to innovations like clock gating and aggressive power management, the CPU no longer consumes a large amount of power while idle. For modern processors the idle power is only a few Watt, while the TDP can go up to 100 Watt. Nowadays the idle power consumers are the motherboard, memory, disks and other peripherals. This leaves ample room for further research, both on the hardware and software side. Software innovations include optimizing the use of available low-power states according to the type of application (background or interactive application) to achieve more energy proportionality. Improving the power management in hardware by making resurrection from low-power states faster, would make it easier to use these states effectively.

In Chapter 5 we measured the energy efficiency of a set of benchmarks and concluded that the most energy-efficient platform for a particular workload depends on its characteristics. In reality we would have a set of workloads and a set of machines, and want to minimize the amount of energy consumed while delivering the results within the deadline. For interactive jobs this deadline can be very strict, in order to serve customers

as fast as possible. For background jobs this deadline can be more relaxed.

To determine the appropriate platform for each of the workloads, a full exploration — running all workloads on all available platforms — might not be feasible. A potential avenue for future work is estimating the appropriate platform while running the benchmark on a single platform. This could be achieved as follows. The first phase is generating a model of the execution time and energy usage for each type of application on each platform. A set of predefined benchmarks could be used to model a wide range of applications. By executing these benchmarks on the available platforms while measuring performance counters, one could build a model that maps the application characteristics (based on the performance counter values) onto the performance metrics (execution time and energy usage). In production, a workload would get scheduled on any available platform. While running the workload we measure its characteristics using hardware performance counters and use the model to check whether this platform is appropriate for this workload or whether the workload should be migrated. This data can then be used to schedule future jobs for the same workload on the appropriate platform.

7.2.3 Analysis of production systems

The TPA language is very powerful: it enables an expert to encode his/her knowledge into formulas that can be used to search for certain behaviors in a trace or set of traces of events. When a formula is matched in a trace, the matcher shows where the formula was matched. For future work we could implement a visualizer that shows the events that are involved in a pattern match. This would enable a performance expert to get even better insight in the exact behavior of the trace.

This visualizer could also be used in a more interactive way as follows. First the performance analyst points to interesting events and interactions between events. In the meanwhile, a TPA pattern is generated in the background. Then all occurrences of the pattern are automatically searched in the trace. This enables the performance analyst to iteratively create patterns and immediately check for occurrences of that pattern. TPA would then serve as the backend for searching and analyzing large traces.

At the moment TPA is primarily used to analyze kernel traces and RPC traces, as shown in Chapter 6. TPA is however not limited to these types of traces and can be used to match a sequence of events in any type of trace. Even an instruction stream can be viewed as a trace of events and TPA could be used to find certain sequences of instructions that cause performance problems at the microarchitectural level. TPA could be integrated with the interval simulator of Chapter 2 to achieve this goal.

Bibliography

[1] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2009.

[2] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *IEEE Computer*, 40:85–87, September 2007.

[3] M. Mayer. What google knows. In *Web 2.0 summit*, 2006.

[4] C. Belady. The green grid datacenter power efficiency metrics: Pue and dcie. http://www.thegreengrid.org/Global/Content/white-papers/The-Green-Grid-Data-Center-Power-Efficiency-Metrics-PUE-and-DCiE, 2007.

[5] Google. Efficiency: How we do it. http://www.google.com/about/datacenters/efficiency/internal/.

[6] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2), May 2009.

[7] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, January 2010.

[8] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: Infrastructure for full system simulation. *SIGOPS Operating System Review*, 43(1):52–61, January 2009.

[9] Energybench v1.0 power/energy benchmarks. http://www.eembc.org/benchmark/power_sl.php.

[10] K.-D. Lange. Identifying shades of green: The SPECpower benchmarks. *IEEE Computer*, 42(3):95–97, March 2009.

[11] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: A balanced energy-efficiency benchmark. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 365–374, June 2007.

[12] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March 2007.

[13] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[14] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 266–277, July 2001.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, October 2008.

[16] D. A. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC)*, pages 163–173, October 2005.

[17] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual IEEE/ACM Symposium on Microarchitecture (MICRO)*, pages 330–335, December 1997.

[18] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 468–477, October 1996.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, October 2002.

[20] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 84–95, June 2003.

[21] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[22] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, March 2007.

[23] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, July 2006.

[24] Ayose Falcón, Paolo Faraboschi, and Daniel Ortega. Combining simulation and virtualization through dynamic sampling. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 72–83, April 2007.

[25] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1 –12, November 2011.

[26] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 295–306, January 2010.

[27] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[28] B. Cmelik and D. Keppel. SHADE: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[29] E. Witchell and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 68–79, June 1996.

[30] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, February 2002.

[31] M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, November 2005.

[32] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[33] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[34] J. Chen, M. Annavaram, and M. Dubois. SlackSim: A platform for parallel simulation of CMPs on CMPs. *ACM SIGARCH Computer Architecture News*, 37(2):20–29, May 2009.

[35] A. Falcón, P. Faraboschi, and D. Ortega. An adaptive synchronization technique for parallel simulation of networked clusters. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 22–31, April 2008.

[36] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 309–322, December 2008.

[37] T. R. Halfhill. Intel's tiny Atom. *Microprocessor Report*, 22:1–13, April 2008.

[38] W. D. Norcott. IOzone filesystem benchmark. http://www.iozone.org/.

[39] A. Avetisyan, R. Campbell, I. Gupta, M. Heath, S. Ko, G. Ganger, M. Kozuch, D. OHallaron, M. Kunze, T. Kwan, K. Lai, M. Lyons, D. Milojicic, H. Y. Lee, Y. C. Soh, N. K. Ming, J. Y. Luke, and H. H. Namgoong. Open cirrus: A global cloud computing testbed. *IEEE Computer*, 43(4):42–50, April 2010.

[40] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. A. Patterson. Rain: A workload generation toolkit for cloud computing applications. Technical Report UCB/EECS-2010-14, Electrical Engineering and Computer Sciences, University of California at Berkeley, February 2010.

[41] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(1):78–103, January 1997.

[42] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: Time-warped network emulation. In *Proceedings of the International Symposium on Networked Systems Design and Implementation (NSDI)*, pages 87–100, May 2006.

[43] P. Ranganathan and P. Leech. Simulating complex enterprise workloads using utilization traces. In *Proceedings of the Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, February 2007.

[44] D. Weisner and T. F. Wenisch. Stochastic queuing simulation for data center workloads. In *Proceedings of the Workshop on Exascale Evaluation and Research Techniques (EXERT)*, March 2010.

[45] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 13–23, June 2007.

[46] L. A. Barroso and U. Hölzle. The case for energy-proportional systems. *IEEE Computer*, 40:33–37, December 2007.

[47] Y. Watanabe, J. D. Davis, and D. A. Wood. WIDGET: Wisconsin decoupled grid execution tiles. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 2–13, June 2010.

[48] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 338–347, June 2010.

[49] J. Guerra, W. Belluomini, J. Glider, K. Gupta, and H. Pucha. Energy proportionality for storage: Impact and feasibility. In *Proceedings of the SOSP Workshop of Hot Topics in Storage and File Systems (HotStorage)*, pages 35–39, October 2009.

[50] D. Meisner, B. T. Gold, and T. Wenisch. PowerNap: Eliminating server idle power. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating ystems (ASPLOS)*, pages 205–216, March 2009.

[51] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz. Towards energy-proportional datacenter memory with mobile dram. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 37–48, June 2012.

[52] D. Wong and M. Annavaram. Scaling the energy proportionality wall through server-level heterogeneity. In *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2012.

[53] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[54] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[55] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, Sept/Oct 2003.

[56] R. Bell, Jr. and L. K. John. Improved automatic testcase synthesis for performance model validation. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS)*, pages 111–120, June 2005.

[57] C. Hsieh and M. Pedram. Micro-processor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1080–1089, November 1998.

[58] A. M. Joshi, L. Eeckhout, R. Bell, Jr., and L. K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(2), August 2008.

[59] A. M. Joshi, L. Eeckhout, L. K. John, and C. Isen. Automated microprocessor stressmark generation. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 229–239, February 2008.

[60] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/. University of Virginia.

[61] V. Vasudevan, D. Andersen, M. Kaminsky, L. Tan, J. Franklin, and I. Moraru. Energy-efficient cluster computing with FAWN: Workloads and implications. In *Proceedings of the First International Conference on Energy-Efficient Computing and Networking (e-Energy)*, April 2010.

[62] J. Moore, J. Chase, K. Farkas, and P. Ranganathan. Data center workload monitoring, analysis and emulation. In *Proceedings of the Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), held in conjunction with HPCA*, February 2005.

[63] D. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, November/December 2000.

[64] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[65] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, pages 93–104, December 2003.

[66] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, December 2009.

[67] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, October 1977.

[68] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC, pages 15–28, 2004.

[69] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[70] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Comput. Syst.*, 13:1–31, February 1995.

[71] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.

[72] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 412–416, 2001.

[73] C. Brzoska. Temporal logic programming with metric and past operators. In *Proceedings of the Workshop on Executable Modal and Temporal Logics*, pages 21–39, 1995.

[74] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, 2006.

[75] http://en.wikipedia.org/wiki/Thread_pool_pattern.

[76] B. Finkbeiner, S. Sankaranarayanan, and H. B. Sipma. Collecting statistics over runtime executions. In *In Proceedings of Runtime Verification (RV02) [1*, pages 36–55, 2002.

[77] H. T. De Beer and B. F. Van Dongen. Process mining and verification of properties: An approach based on temporal logic. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005, volume 3760 of Lecture Notes in Computer Science*, pages 130–147, 2005.

[78] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, 1982.

[79] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[80] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *J. Syst. Softw.*, 12:107–123, May 1990.

[81] G. S. Goldszmidt, S. Yemini, and S. Katz. High-level language debugging for concurrent programs. *ACM Trans. Comput. Syst.*, 8:311–336, November 1990.

[82] R. T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic*. 1982.

[83] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *Computer*, 18:10–19, February 1985.

[84] K. Trentelman and M. Huisman. Extending jml specifications with temporal logic. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, AMAST '02, pages 334–348, 2002.

[85] A. Viswanathan, A. Hussain, J. Mirkovic, S. Schwab, and J. Wroclawski. A semantic framework for data analysis in networked systems. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 10–10, March 2011.

[86] M. Frey and A. Weininger. Using temporal logic specifications to debug parallel programs. *Microprocess. Microprogram.*, 39:97–100, December 1993.

[87] P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28(110-111):119–136, 1985.

[88] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374. www.cidrdb.org, 2007.

[89] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1100–1102, June 2007.

[90] G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, July 2010.

[91] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 193–206, June 2009.

[92] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, June 2006.

[93] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 365–383, October 2005.

[94] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 385–402, October 2005.

[95] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*, EuroPVM/MPI'06, pages 303–312, September 2006.

[96] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 739–753, October 2010.

[97] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 117–132, October 2009.